PARALLEL ALGORITHMS FOR LARGE-SCALE COMPUTATIONAL METAGENOMICS

By

CHANGJUN WU

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
Department of Electrical Engineering and Computer Science

MAY 2011

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of CHANGJUN WU find it satisfactory and recommend that it be accepted.

Ananth Kalyanaraman, Ph.D., Chair

Shira L. Broschat, Ph.D.

William R. Cannon, Ph.D.

Diane J. Cook, Ph.D.

Amit Dhingra, Ph.D.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Ananth Kalyanaraman for his professional guidance and supervision during my Ph.D. studies. His passion about research always inspires me in my research. When I was confused about my research, he is always there ready to help me out. I am so grateful for all the weekends and late nights he sacrificed for me. This dissertation would not be possible without his supervision and guidance. Among several of his amazing qualities, his quick way of thinking, clear way of presenting, creative way of researching and critical way of reasoning are truly a bless to our students. He is such a amazing teacher and researcher in all means, and really sets a wonderful example in the rest of my career.

Many thanks to my committee members Dr. Shira Broschat, Dr. William Cannon, Dr. Diane Cook, and Dr. Amit Dhingra for their insightful advices and guidances in my research. Specifically, Dr. Broschat always encourages and supports me in my research, and I benefited tremendously through insightful conversations we had. Dr. Cannon's extensive knowledge on both computer science and biology helps me tremendously understand the state-of-the-art research in Bioinformatics during my summer internship in Pacific Northwest National Laboratory. The creative and

passionate teaching style of Dr. Cook always amazed me, and her pioneering way of conducting research set an amazing example for my future career. Without Dr. Amit Dhingra I would not be able to step into the area of bioinformatics smoothly; his vision about bioinformatics research always sparkles me, and he is such a great teacher anytime I was confused by the biological concepts.

In addition, I would also like to thank Dr. K.C. Wang for his amazing operating system course, which really opened my mind and helped me significantly in my research. He is such an expert on teaching and researching on operating system. He always can explain the most difficult concepts and algorithms in such an understandable way. The knowledge and skills he has is a true bless to our students.

This research was primarily supported by NSF Grant – IIS 0916463, and in part by WSU Office of Research.

I am also grateful to my beloved families during these years of studies; nothing would be possible without their financial and mental support. I would also like to thank all the members in our research lab: Gaurav Kulkarni, Vandhana Krishnan, Md. Muksitul Haque, Benjamin Latt, Hao Lu, Souradip Sarkar, Meenakshi Rameshkumar and Inna Rytsareva; without the company of you, my life in Pullman would not be as colorful as it is now!

PARALLEL ALGORITHMS FOR LARGE-SCALE COMPUTATIONAL METAGENOMICS

Abstract

by Changjun Wu, Ph.D.
Washington State University
May 2011

Chair: Ananth Kalyanaraman

Developing high performance computing solutions for modern day biological problems present a unique set of challenges. The field is experiencing a data revolution due to a rapid introduction of several disruptive experimental technologies. Consequently, computational methods that analyze biological data are currently being put to the test in their capability to scale to massive data sizes. Added to this data-intensiveness, is the brand of computation that is quite different in flavor to that in other, perhaps more traditional scientific computing fields. The problems are dominated by integer arithmetic, string matching, combinatorial space exploration, and graph-theoretic formulations that introduce irregularity in computation and communication patterns.

In this dissertation, we report on our efforts to bridge the gap between biological data processing and high performance computing solutions. Specifically, we focus on the problem of clustering very large collections of amino acid sequences on distributed memory supercomputers. Given a set of amino acid sequences we reduce the problem to one of constructing sequence homology graph and subsequently detecting arbitrarily-sized dense subgraphs. Our approach efficiently parallelizes this task on a distributed memory machine through a combination of divide-and-conquer and combinatorial pattern matching heuristic techniques. The major algorithm/software contributions of this dissertation are: (1) *pGraph*: for parallel construction of sequence homology graph; and (2) *pClust*: for graph-based sequence clustering. Preliminary tests on an arbitrary collection of $\sim$2 million amino acid sequences from the Global Ocean Sampling project database reveal that our new approach is able to improve sensitivity, recruit more sequences, while considerably reducing the time to solution and memory requirement. The algorithmic techniques developed as part of this research have a wider applicability to other applications in computational biology wherever the need for conducting large-scale sequence analysis is the primary bottleneck.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Dedication

dedicated to my beloved parents!

CHAPTER 1. INTRODUCTION

Whatever you can do, or dream you can, begin it. Boldness has genius, power, magic in it.

— Goethe

Metagenomics (also called community genomics or environmental genomics) [Handelsman et al., 1998] is the field resulting from the application of modern genomics techniques to study environmental microorganisms directly in their natural environments. In traditional microbiology, microbes are isolated from their natural environment and subsequently cultured in laboratory for further studies. However such cultivable microbes only account for a very small fraction ($< 1\%$) of the microbes inhabiting in the natural environments; other microbes that inhabit environments (e.g. the deep sea vent, soil, human intestine, and hot springs) live as a community and cannot be cultured individually under laboratory settings. Such microbial communities from environmental habits play a vital role in their environments. For example, microbes in the soil help plants to absorb nitrogen and minerals; microbes in the human gut help us to break down toxin and digest food; microbes in the ocean help to clean up the pollutants. Metagenomics aims to study such microbial commu-

nities living in various environmental habitats. Among several applications, metagenomics is expected to create high impact on bioenergy, environmental biotechnology, pharmaceuticals and agriculture [Handelsman, 2004]. Recent advances in genomics technologies are beginning to enable the study of these environmental microbes, and provide both structural and functional insights into the microbial communities that inhabit these environments.

Metagenomics sequencing starts with deoxyribonucleic acid (DNA) extraction from a target environment, and the DNAs are cloned into vectors subsequently. These clones are then transformed into a host bacterium, and subsequently sequenced using a random shotgun strategy. In this strategy, the genomes are cut into *fragments* at random sites, and each fragment is subsequently fed into a DNA sequencing machine in order to determine the composition of the sequence. Thereafter, the sequences fragments (*reads*) are computationally assembled into contig sequences. Once sequenced and assembled, the data can help us answer two fundamental questions about the target environmental microbial community: "who all are out there?" and "what are they doing?" While assembling can partially help in answering the first question, the second question poses more interesting challenges and will be the focal point of this dissertation. *More specifically, our goal is to develop computational methods to characterize the proteins represented in metagenomics communities.*

Proteins are molecules responsible for performing most of the cellular functions

in an organism. Proteins that are evolutionarily- (and thereby functionally-) related are said to belong to the same "family". Identifying protein families is of fundamental importance to document the diversity of the known protein universe. It also provides a means to determine the functional roles of newly discovered protein sequences. This latter cause has become highly significant of late because numerous genome projects have been completed and as a result there is a sudden expansion of the protein universe. The most dominant contributor to this information revolution has been the projects in metagenomics.

Over the last couple of years, numerous metagenomics projects have been initiated – [Gill et al., 2006, Noguchi et al., 2006, Rusch et al., 2007, Schleper et al., 2005, Tyson et al., 2004, Venter et al., 2004, Yooseph et al., 2007] to name a few. A continued analysis of their DNA pool is leading to collections of millions of new amino acid sequences (called Open Reading Frames – ORFs)[1] with a potential to be functional proteins. Therefore, if the ORF data can be organized into functionally related groups then it would shed vital insights into community functions. Here, two types of inferences can be made. If a newly derived ORF can be mapped to known protein family, then its functional role can be determined based on the known

[1]Henceforth, we use the terms "ORFs" and "amino acid sequences" interchangeably thereafter in this dissertation.

protein family. Otherwise, groups of ORFs that share similarity with one another but not with any sequences in known protein families can lead to the inference of *de novo* protein families. Either way, the resulting inferences can significantly enrich our understanding of the protein universe represented in the underlying metagenomics communities.

1.1 Challenges in metagenomics data analysis

In bioinformatics, and particularly in metagenomics, the development of computational algorithms and tools has been unable to keep up with the fast growing data. Given the number of past and ongoing metagenomics projects, thousands of new sequences are produced on a daily basis. As of 2007, the Global Ocean Sampling (GOS) project [Yooseph et al., 2007] alone deposited ∼17 million new sequences, which exceeded the number of all known microbial proteins in public repositories, such as the NCBI GenBank [The national center for biotechnology information., 2011], UniprotKB/Swiss-Prot [Swiss institute of bioinformatics., 2011], and IMG/M [Markowitz et al., 2008]). This explosive expansion renders the existing suite of algorithms and tools inefficient to be adopted into practice.

Current approaches for detecting protein families operate by computing all-versus-all pairwise sequence similarity, and subsequently using heuristic techniques to

deduce family relationships from pairwise similarities. There are two key limitations to this approach: (1) Computing all-versus-all pairwise similarities could become computationally prohibitive even for hundreds of thousands of sequences because for n sequences the run-time complexity is $\Omega(n^2)$. This high complexity is often times offset by compromising the quality of the output; and (2) The algorithms to deduce family relationships from pairwise similarities *store* all pairwise results making the space complexity $\Theta(n^2)$. While such high complexity in time and space should make the problem an ideal candidate to benefit from parallel computing, there are hardly any parallel approaches. Even those that deploy parallelism resort to brute-force allocation of tasks across multiple computers and to using specialized large-memory high-end platforms for tackling the space problem. For example, a recent analysis [Yooseph et al., 2007] of ∼28.6 million ORFs took an aggregate 10^6 CPU hours. The task was parallelized using 125 dual processors systems and 128 16-processor nodes each containing between 16GB-64GB of RAM.

In addition to the large data size, there are other computational challenges to contend with in metagenomics sequence analysis.

1. **Data intensiveness:** Current approaches use fast, heuristic methods (e.g. BLAST [Altschul et al., 1990]) to detect sequence homology. For better accuracy, optimality guaranteeing dynamic programming methods need to be used. However, running such methods for billions of sequence pairs could become run-

time prohibitive if executed inefficiently. In our experiments, an all-against-all pairwise sequence comparison of 20K sequences using dynamic programming runs for at least 40 CPU hours.

2. **Data locality:** A second issue is data locality (ensuring data availability during computation) becomes a serious problem for parallel processing with the rapidly growing sequence inputs, because of the limited memory available on an individual machine.

3. **Irregularity:** A third challenge is specific to amino acid sequence inputs from metagenomics data. Compared to DNA sequences, the length distribution of amino acid sequences in metagenomics datasets is *nonuniform*, leading to irregular processing time for each pair of sequences. As a concrete example, an all-against-all comparison analysis of 28 million human DNA sequences in the 2001 Human genome project [Venter et al., 2001] consumed 10^4 CPU hours; whereas 10^6 CPU hours were required to process roughly the same number of metagenomics ORFs in 2007 [Yooseph et al., 2007] despite the obvious use of much faster machines.

1.2 Contributions

In this dissertation, we present the design and development of parallel algorithms and software for characterizing large-scale data sets of amino acid sequences derived from environmental microbial communities. More specifically, we treat the problem of protein family characterization as one of graph-theoretic clustering and propose algorithmic and software solutions that can exploit the aggregate memory and compute power of massively parallel distributed memory supercomputers.

The major contributions in this dissertation are as follows:

1. **pGraph:** First, we present a scalable algorithm to detect pairwise amino acid sequence homology on large-scale distributed memory supercomputers. The input is a set of amino acid sequences and the output is a sequence homology graph. Our approach is unique in two ways: (1) It adequately addresses the inherent irregularities that originate during parallel computation of the sequence homology graph; and (2) It makes the computation of the large graphs using the more sensitive optimality-guaranteeing dynamic programming methods feasible in practice. Experimental results shows that our method scales linearly up to thousands of processors on large metagenomics collections.

2. **pClust:** Next, we present an algorithmic heuristic to compute the core of se-

quence clusters using the graph constructed at the first stage. More specifically, we treat the problem as one of finding densely connected subgraphs within large-scale graphs. Our method is different from other existing methods in that it uses a more accurate heuristic that was originally developed for web community detection on internet data. By transforming this heuristic to work for amino acid sequence clustering, we show that it is possible to achieve significant improvements in the quality of the output. Furthermore, we developed several algorithmic techniques that result in significant performance improvements as well.

Extensive experimental evaluation of the algorithms was conducted on state-of-the-art supercomputers, and the results demonstrate the overall effectiveness of our approach in its ability to scale to millions of sequences on thousands of processors, while producing outputs that are qualitatively better than those produced by other methods. All implementations have been made available as open source.

It is to be noted that the ideas originating in this dissertation can be applied to a broader segment of applications outside protein family characterization. The tool for graph clustering, *pClust*, is a generic dense subgraph detection tool that can work on arbitrary graphs. Similarly, while our implementation of *pGraph* is specific to building amino acid sequence homology detection, the main ideas in parallelization can be carried forward to other data-intensive scientific applications where irregularities in

computation patterns pose challenges for scalability.

1.3 Dissertation organization

The dissertation is organized as follows. Chapter 2 presents the basic biological background and computational tools used in this dissertation. Chapter 3 presents the design and development of our scalable parallel algorithm (*pGraph*) for sequence homology detection, alongside performance studies on large-scale supercomputers. In Chapter 4, we transform the problem of sequence clustering into one of dense subgraph detection problem; following which, our solution called (*pClust*) was evaluated extensively on real world data sets with performance and qualitative studies. Chapter 5 concludes the dissertation with a discussion on prospective research directions.

CHAPTER 2. BACKGROUND

Nothing great was ever achieved without enthusiasm.

— Emerson.

2.1 Biological background

DNA and Genome: Deoxyribonucleic acid (DNA) is a double stranded molecule that constitutes the genetic material for most living organisms. Each DNA is made up of a sequence of smaller molecules called *"nucleotides"*. There are four different types of nucleotide bases - Adenine (abbreviated A), Cytosine (C), Guanine (G) and Thymine (T). The sequences in the two strands of a DNA are related through a *"base complementary"* rule which is defined as follows: *"A"* pairing with *"T"*, *"C"* pairing with *"G"*. By convention only one strand of the DNA sequence is listed as the representative of the two strands, as the other strand can be deduced using the complementary pairing rule. Also every DNA sequence is associated with a direction which starts from 5' to 3' end. The set of all DNA in any cell of a living organism is defined as the organism's *genome*.

RNA, Protein, Gene, and ORF: Similar to the DNA, an RNA is also a chain of nucleotides. The only exceptions are: (1) instead of Thymine (T), Uracil (U) is found; and (2) an RNA is a single stranded molecule. For a protein sequence, the basic construction units are *"amino acids"*, so it is also called an "amino acid sequence". There are 20 known amino acids; they are connected to each other through peptide bonds. *Genes* are subsequences in a DNA molecule that code for an RNA molecule (a process known as *transcription*). An RNA molecule can then be translated into a corresponding protein product based on the genetic code (a process known as *translation*). In eukaryotic cells, only parts (called *"exons"*) of each gene are transcended into RNA molecule; the remaining parts are referred to as *"introns"* (no-coding regions), and this process is known as "splicing". In prokaryotic cells, however a gene does not contain intronic regions; for the translation, every three consecutive nucleotides (called a *"codon"* are translated into an amino acid according to the genetic code. Given an RNA sequence without the location information of the stop codon, there are six possible ways to translate the sequences, and each translation produces what is known as an *Open Reading Frame* or *ORF*. The ORFs represent putative candidates of the real protein sequences.

DNA Sequencing and Assembly: DNA sequencing is a process of determining the nucleotide composition of a DNA molecule. Most popular technologies include the traditional Sanger sequencing [Sanger et al., 1977], and a plethora of next-

generation technologies such as 454 [454 life scientces., 2011], Illumina (Solexa) [Schuster, 2008], and SOLiD [Abi solid sequencing., 2011]. Nevertheless, they can only be directly applied to small fragment of sequences. Therefore to determine the DNA composition of a long sequence, shotgun sequencing is employed in recent sequencing projects. In the shotgun sequencing process, the parent DNA is first randomly cut into small fragments; then each fragment is fed into a sequencer to have its sequence determined. Finally, the sequenced fragments are pieced together to assemble the sequence of the parent DNA through a computational process called *"genome assembly"*.

2.2 Computational tools

Pairwise sequence alignment: The most effective way to study the evolutionary relationship between a pair of sequences is by examining their sequence homology. Two sequences are said to be *"homologous"* if they show a "sufficiently" high degree of sequence similarity. There exists several standard dynamic programming algorithms [Gotoh, 1982, Needleman and Wunsch, 1970, Smith and Waterman, 1981] to find the optimal similarity between two sequences. For example, the Needleman-Wunsch algorithm is used for computing optimal global alignment and the Smith-Waterman algorithm for optimal local alignment. All these dynamic programming

based algorithms run in time proportional to the product of the lengths of the two sequences, i.e. $\Theta(mn)$ for two sequences of length m and n. Despite the optimality of dynamic programming, it can be prohibitive slow when comparing billions of pairs of sequences. For instance, if we need to search a sequence against a database, which is represented by concatenating all the sequences in it, then the run-time will be prohibitively expensive as the time complexity of dynamic programming is the product of the total sequence lengths in database and the length of the query sequence

BLAST (basic local alignment search tool): To speed up this process, a heuristic method, called "Basic Local Alignment Search Tool" [Altschul et al., 1990] (BLAST) is widely used by the bioinformatics community despite its relative low sensitivity [Shpaer et al., 1996]. BLAST can perform sequence comparison for both amino acid (protein) sequence and nucleotide (DNA) sequence. The main idea behind BLAST is "seed-and-extend": First, fixed-length short matches, called "k-mers" from both sequences are partitioned into corresponding buckets based on their similarity score; thereafter the short regions with high similarity score are identified as "seeds". During the "extend" phrase, the matched regions are extended in both directions while accounting for mismatches or gaps until their alignment scores drop below a predefined cutoff. Finally, the extended regions are chained diagonally, and each individual chained region is reported as a "hit" in the final alignment output. Comparing to the dynamic programming approaches, the final alignment output by

BLAST is not guaranteed to be optimal.

Parallel versions of BLAST: Despite the high efficiency of the BLAST algorithm, performance could still become an issue for large databases. For instance, an all-against-all comparison of 20K sequences using BLAST consumes over 15 CPU hours on a state-of-the-art computer. In order to keep up with the explosive sequence growth, two parallel versions of BLAST, scalaBLAST from [Oehmen and Nieplocha, 2006] and mpiBLAST from [Darling et al., 2003] are available. Fundamentally, the serial version of BLAST serves as the cornerstone in these parallel approaches, and performance is their primary motivation. Nevertheless, sensitivity is becoming a significant concern of late, especially when dealing with highly fragmented metagenomics data. It is more desirable to use the dynamic programming algorithms to compute the homology between a pair of sequences.

Suffix tree: Suffix tree [Weiner, 1973] is a compacted *trie* of all suffixes in a given string. The edges in the suffix tree are labeled with substrings of the input string, such that each suffix is expressed by a path starting from the root to a leaf in the tree. Figure 2.1 shows a suffix tree example for string *banana*. The construction of the suffix tree for a string can be achieved in a linear time and space complexity [McCreight, 1976, Ukkonen, 1995, Weiner, 1973]. Suffix trees have direct applications in a number of matching problems. Examples of such applications include pattern matching, finding the longest repeated substring, finding the longest

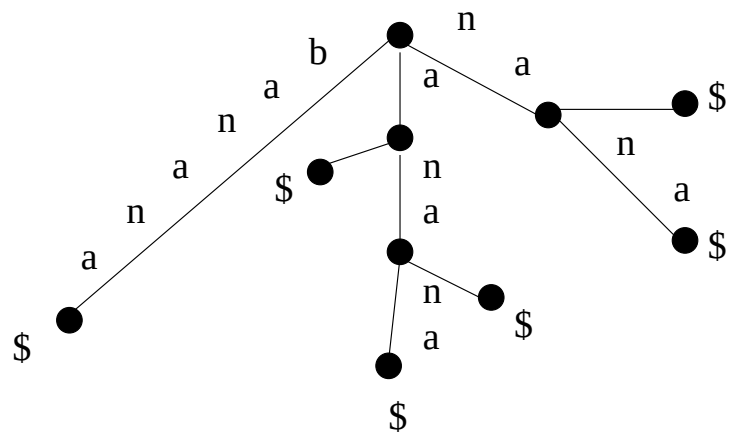

Figure 2.1: A suffix tree example for string *banana*. A delimiter $ is appended to the end of each suffix.

common substring, and finding the longest palindrome in a string. A suffix tree is called a *Generalized Suffix Tree* (GST) if it is constructed for a set of sequences. For each leaf in GST, a sequence identification information is contained accordingly to differentiate the suffixes coming from different sequences. To keep up with the rapid data expansion, a couple of parallel algorithms have been developed [Kalyanaraman et al., 2007, Ghoting and Makarychev., 2009].

CHAPTER 3. SEQUENCE HOMOLOGY DETECTION

Those are my principles. If you don't like them, I have others.

— Groucho Marx

3.1 Introduction

Protein sequence homology detection is a fundamental problem in computational molecular biology. Given a set of protein sequences, the goal is to identify all pairs of homologous sequences, where similarity constraints are typically defined by user-specified parameters under an alignment model (e.g., [Needleman and Wunsch, 1970, Smith and Waterman, 1981]). In graph-theoretic terms, the protein sequence homology detection problem is one of constructing an undirected graph $G(V, E)$, where V is the set of input protein sequences and E is the set of edges (v_i, v_j) such that the sequences corresponding to v_i and v_j are highly similar.

Most algorithms in bioinformatics that process sequence homology graphs assume that the graph can be easily built or is readily available as input. However, modern-day use-cases suggest otherwise. Large-scale projects generate millions of *new* sequences that need to be matched against themselves and against sequences al-

ready available and consolidated from previous sequencing projects. For example, the ocean metagenomics project generated more than 17 million new sequences and this set was analyzed alongside 11 million sequences in public protein sequence databanks (for a total of 28.6 million sequences). Consequently, the most time consuming step during analysis was homology detection, which alone accounted for 10^6 CPU hours despite the use of faster approximation heuristics such as BLAST [Altschul et al., 1990] to determine homology. Ideally, dynamic programming algorithms [Needleman and Wunsch, 1970, Smith and Waterman, 1981] that guarantee alignment optimality should be the method of choice as they are generally more sensitive but the associated high cost of computation coupled with a lack of support in software for coarse-level parallelism have impeded their application under large-scale settings.

In this chapter, we address the problem of parallelizing homology graph construction on massive protein sequence data sets, and one that will enable the deployment of the optimality-guaranteeing dynamic programming algorithms as the basis for pairwise homology detection (or equivalently, edge detection). Although at the offset the problem may appear embarrassingly parallel (because the evaluation of each edge is an independent task), several practical considerations and our own experience [Wu and Kalyanaraman., 2008] suggest it is a non-trivial problem.

Firstly, the problem is data-intensive, even more so than its DNA counterpart. While the known protein universe is relatively small, modern use-cases particularly in

metagenomics, in an attempt to find new proteins and families, generate millions of DNA sequences first and then convert them into amino acid sequences corresponding to all six open reading frames as protein candidates for evaluation, resulting in a 6× increase in data volume for analysis[2]. Tens of millions of such amino acid sequences are already available from public microbial repositories (e.g., CAMERA [CAMERA, 2011], IMG/M [Markowitz et al., 2008]), and further accumulation is expected. Large data size creates two complications.

1. A brute-force all-against-all sequence comparison strategy to detect the presence of edges becomes practically unfeasible due to the quadratic explosion in the number of alignment computations. Instead, a filtering based strategy, one that short-lists a smaller subset of sequence pairs based on their potential to pass the alignment criteria prior to actually computing the alignments, needs to be used. In practice, exact matching filters that deploy string data structures such as suffix trees [Weiner, 1973] are highly effective in reducing alignment work without compromising on quality [Kalyanaraman et al., 2003b, 2007]. While the time consumed by these advanced filters for pair generation is relatively less when compared to alignment computation, it is certainly *not negligible*.

[2]Henceforth for simplicity of exposition, we will use the terms "amino acid sequences", "protein sequences" and "ORFs" interchangeably; although in practice an amino acid sequence need not represent a complete or real protein sequence.

From a parallel implementation standpoint, this means that we could not use a standard work distribution tool — instead, work generation also needs to be parallelized dynamically alongside work processing, in order to take advantage of these sophisticated filters.

2. A large data size also means that the local availability of sequences during alignment processing cannot be guaranteed under the distributed memory machine setting. Alternatively, moving computation to data is also virtually impossible because a pair identified for alignment work could involve arbitrary sequences and could appear in an arbitrary order during generation, both of which are totally data-dependent.

A second major challenge in protein sequence homology detection is that the handling of amino acid sequence data gives rise to some unique *irregularity* issues that need to be contended with during parallelization.

1. Assuming *"work"* refers to a pair of sequences designated for alignment computation, *the time to process each unit of work could be highly variable.* This is because the time for aligning two sequences using dynamic programming takes time proportional to the product of the lengths of the two sequences [Needleman and Wunsch, 1970, Smith and Waterman, 1981]. And, amino acid sequences tend to have a substantial variability in their lengths, as we will also shown in

Section 4.4.

2. For amino acid data, *the rate at which work is generated could also be non-uniform.* For instance, similar sized portions of the suffix tree index could yield drastically different number of sequence pairs, as the composition of the index is data dependent. *A priori* stocking of pairs that require alignment is simply not an option because of a worst-case quadratic requirement.

Note that these challenges do not typically arise when dealing with DNA. For instance, in genome sequencing projects the lengths of raw DNA sequences derived from modern day DNA sequencers are typically uniform. This coupled with the nature of sampling typically leads to predictable workload during generation and processing. In case of metagenomics protein data, the higher variability in sequence lengths is a result of the translation done on the assembled products of DNA assembly (i.e., not raw DNA sequences). Because of this variability, analysis of protein data tends to take longer time and more difficult to parallelize. For example, in the human genome assembly project [Venter et al., 2001], the all-against-all sequence homology detection of roughly 28 million DNA sequences consumed only 10^4 CPU hours. Contrast this with the 10^6 CPU hours observed for analyzing roughly the same number of protein sequences in the ocean metagenomic project despite the use of much faster hardware [Yooseph et al., 2007].

The rest of this chapter is organized as follows. Section 4.2 presents the current state of art for parallel sequence homology detection. Section 4.3 presents our proposed method and implementation details. Experimental results are presented and discussed in Section 4.4, and Section 3.5 concludes this chapter.

3.2 Related work

Sequence homology between two biomolecular sequences can be evaluated either using rigorous optimal alignment algorithms in time proportional to product of the sequence lengths [Needleman and Wunsch, 1970, Smith and Waterman, 1981], or using faster, approximation heuristic methods such as BLAST [Altschul et al., 1990] and FASTA [Pearson and Lipman., 1988]. Detecting the presence or absence of pairwise homology for a *set* of protein/amino acid sequences, which is the subject of this chapter, can be modeled as a homology graph construction problem with numerous applications (e.g., [Apweiler et al., 2004, Bateman et al., 2004, Enright et al., 2002, Kriventseva et al., 2001, Olman et al., 2007]). The rapid adoption of cost-effective, high throughput sequencing technologies is contributing millions of new sequences into sequence repositories [CAMERA, 2011, Markowitz et al., 2008, The national center for biotechnology information., 2011]. As a result, detection of pairwise homology over these large data sets is becoming a daunting computational task.

An indirect option for implementing homology detection is to use the NCBI BLAST program [Altschul et al., 1990], which is a method originally designed for performing sequence database search (query vs. database). This can be done by setting both the query and database sets to the input set of sequences. For instance, the ocean metagenomics survey project [Yooseph et al., 2007] used BLAST to perform all-against-all sequence comparison. This took 10^6 CPU hours — a task that was parallelized, albeit in an *ad hoc* manner, by manually partitioning across 125 dual processors systems and 128 16-processor nodes each containing between 16GB-64GB of RAM. Several mature parallel tools are available for BLAST — the most notable tools being mpiBLAST [Darling et al., 2003] and ScalaBLAST [Oehmen and Nieplocha, 2006]. These methods run the serial version of NCBI BLAST at their cores, while offering a high degree of coarse-level parallelism and have demonstrated scaling to high-end parallel machines. In addition to being relatively quicker, BLAST also provides a statistical score of significance for comparing a query sequence against a database of sequences.

The use of BLAST based techniques however comes with reduced sensitivity [Pearson., 1991, Shpaer et al., 1996], as the underlying algorithm is really an approximation heuristic for computing alignments. Comparatively, the dynamic programming algorithms offer alignment optimality but are generally a couple of orders of magnitude slower. Nevertheless, sensitivity is becoming a significant concern of late,

especially when dealing with metagenomics data processing because of its highly fragmented nature of sampling. Another less desirable side effect of using BLAST for protein sequence data is that it uses the lookup table data structure which is limited to detection of only short, fixed-length matches between pairs of sequences. This could result in more pairs to be evaluated. Other string data structures such as suffix trees provide more specificity when it comes to the choice of pairs to evaluate due to their ability to detect longer, variable-length matches.

Due to the advantages in using dynamic programming, there has been a flurry of efforts for implementing hardware-level acceleration for optimal pairwise sequence alignment computation on different architectures (reviewed in [Sarkar et al., 2010]). However, there is a dearth in research that has targeted at achieving coarse-level parallelism for carrying out millions of such alignment computations. There have been a few efforts for DNA sequence analysis [Kalyanaraman et al., 2003a, 2007], but carrying out protein sequence homology detection at a large-scale has not been addressed to the best of our knowledge.

3.3 Algorithms

Notation: Let $S = \{s_1, s_2, \ldots, s_n\}$ denote the set of n input protein sequences. Assuming $|s|$ denotes the length of sequence s and $m = \Sigma_{i=1}^{n}|s_i|$ denotes the total length

of all sequences in S. Let τ be the predefined redundant sequence similarity cutoff. Let $G = (V, E)$ denote a graph defined as $V = S$ and $E = \{(s_i, s_j) \mid s_i$ and s_j are "similar", defined as per pre-defined alignment cutoffs.$\}$. We use the term "pair" to refer to an arbitrary pair of sequences (s_i, s_j).

3.3.1 Suffix tree indexing

A brute-force approach to detect the presence of an edge is to enumerate all possible $(\binom{n}{2})$ pairs of sequences and retain only those as edges which pass the alignment test. Alternatively, since alignments represent approximate matching, the presence of long exact matches can be used as a necessary but not sufficient condition. This approach can filter out a significant fraction of poor quality pairs and thereby reduce the number of pairs to be aligned significantly. Suffix tree based filters provide one of the best filters — for instance, anywhere between 67% to over 99% savings for our experiments shown later in the results section (Table 3.2).

To implement exact matching using suffix trees, we use the optimal pair generation algorithm described in [Kalyanaraman et al., 2003a], which detects and reports all pairs that share a maximal match of a minimum length ψ. The algorithm first builds a Generalized Suffix Tree (GST) data structure as a string index for the strings

in S and then traverses the tree in a bottom-up fashion to generate pairs from different nodes. Suffix tree construction is a well studied problem in both serial and parallel, and any of the standard, serial linear-time construction methods [McCreight, 1976, Ukkonen., 1990, Weiner, 1973] can be used, or efficient distributed memory codes can be used for parallelism [Kalyanaraman et al., 2007, Ghoting and Makarychev., 2009]. Either way, the tree index can be generated in one preprocessing step and stored in the disk[3].

For our purpose, we generate the tree index as a forest of disjoint subtrees emerging at a specified depth $\leq \psi$, so that the individual subtrees can be independently traversed in parallel to generate pairs. Given that the value of ψ is typically a small user-specified constant, the choice for the cutting depth is restricted too. This implies that the size distribution of the resulting subtrees can be *nonuniform* and is input dependent. It is also to be noted that, even though the pair generation algorithm runs in time bound by the number of output pairs, the process of generation itself could also be *nonuniform* — in that, subtrees of similar size could produce different number of pairs and/or at different rates, and the behavior is input-dependent.

[3]Note that there are other, more space-efficient alternatives to suffix trees such as suffix arrays and enhanced suffix arrays, which can also be equivalently used to generate these pairs with some appropriate changes to the pair generation code. That said the type of challenges dealt with the tree during parallel pair generation and the solutions proposed would still carry over to these other representations.

For instance, if a section of subtree receives a highly repetitive fraction of the input sequences then it is bound to generate a disproportionately large number of pairs. Encouragingly, a small value for the cutting depth is *not* a limiting factor when it comes to the number of subtrees and is sufficient to support a high degree of parallelism. This is because the number of subtrees is expected to grow exponentially with the cutting depth; for instance, a cutting depth as small as 4 on a tree built out of protein sequences (alphabet size 20) could produce around $160K$ trees (as shown in experimental results).

3.3.2 Graph construction

Problem 1 *Given a set S of n protein sequences and p processors, the protein sequence graph construction problem is to detect and output the edges of $G = (V, E)$ in parallel.*

We present here an efficient parallel algorithm to construct the homology graph G using the suffix tree index constructed in the previous step and the input sequence set S. The inputs include the sequence set S and the tree index T. The tree index is available as a forest of k subtrees, which we denote as $T = \{t_1, t_2, \ldots t_k\}$. The output of *pGraph* should be the set of all edges of the form (s_i, s_j) s.t., the sequences s_i and s_j

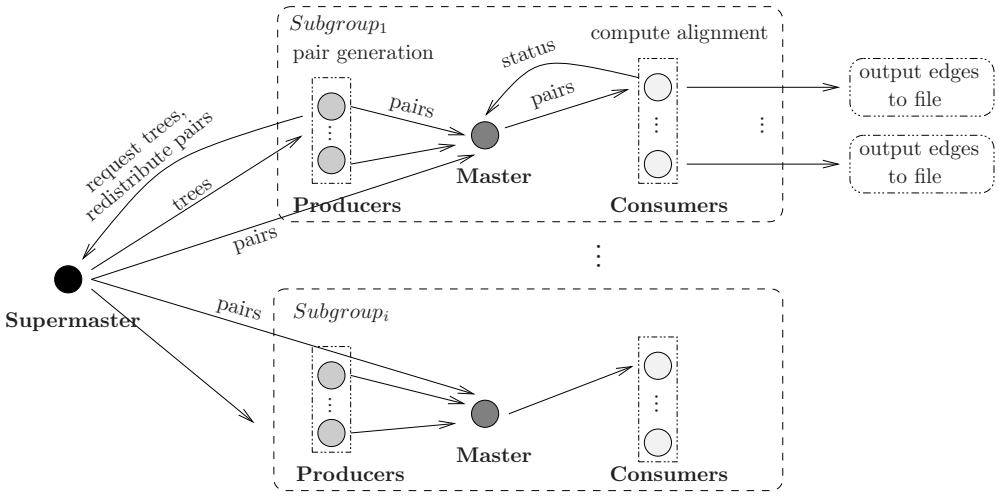

Figure 3.1: The overall system architecture for *pGraph*.

pass the alignment test based on user-defined cutoffs. There are two major operations that need to be performed in parallel: i) generate pairs from the tree index; and ii) compute sequence alignments and output edges if they pass the predefined cutoffs.

Our method uses a hybrid variant between the hierarchical multiple-master/worker model and producer-consumer model to counter the challenges posed by the irregularities in pair generation and alignment rates. The overall system architecture is illustrated in Figure 3.1.

Given p processors and a small number $q \geq 3$, the parallel system is partitioned as follows: i) one processor is designated to act as the *supermaster* for the entire system; and ii) the remaining $p - 1$ processors are partitioned into subgroups of

size q processors each[4]. Furthermore, each subgroup is internally organized with r processors designated to the role of *producers*, one processor to the role of a *master*, and c processors to the role of *consumers*, where $c = q - r - 1$.

At a high level, the producers are responsible for pair generation, the masters for distributing the alignment workload within their respective subgroups, and the consumers for computing alignments. The supermaster plays a supervisory role to ensure load is distributed evenly among subgroups. Unlike traditional models, the overall data flow is from supermaster to the subgroups and also back (for redistribution). In what follows, we describe the various design factors and present algorithms and protocols for each component in the system.

Let:

$P_{buf} \leftarrow$ a fixed sized pair buffer at the producer;

$M_{buf} \leftarrow$ a fixed sized pair buffer at the master;

$C_{buf} \leftarrow$ a fixed sized pair buffer at the consumer;

$S_{buf} \leftarrow$ a fixed sized pair buffer at the supermaster;

$b_1 \leftarrow$ batch size (for pairs) from producer or supermaster to master;

$b_2 \leftarrow$ batch size (for pairs) from master to consumer;

---

[4]With the possible exception of the last subgroup which may obtain less than q processors if $(p-1)\%q \neq 0$.

Producer: the primary responsibility of a producer is to load a subset of subtrees in T and generate pairs using the maximal matching algorithm in [Kalyanaraman et al., 2003a]. The main challenge here is that trees allocated at a producer could result in generation of pairs at a variable rate, although this generation rate is virtually guaranteed to be faster than the rate of consumption (alignment). This is because the pair generation is a simple cross product of sets at any given tree node. To tackle an overactive producer, we maintain a fixed-size pair buffer at each producer ($\sim 80MB$ in our current implementation) and pause the generation process when the buffer is full. This is possible because the pair generation algorithm in [Kalyanaraman et al., 2003a] is an on-demand method. Furthermore, the tree allocation is left to the supermaster and pair allocation from the producer is left to the local master in our design.

More specifically, we follow the algorithm shown in Algorithm 1. Initially, a producer fetches a batch of subtrees (available as a single file) from the supermaster. The producer then starts to generate and enqueue pairs into P_{buf}. Subsequently, the producer dequeues and sends b_1 pairs to the master. This is implemented using a nonblocking send so that when the master is not yet ready to accept pairs, the producer can continue to generate pairs, thereby allowing masking of communication. After processing the current batch of subtrees, the producer repeats the process

by requesting another batch of subtrees from the supermaster. Once there are no more subtrees available, the producers dispatch the rest of pairs to both master *and* supermaster, depending on whoever is responsive to their nonblocking sends. This strategy gives the producer an option of redistributing its pairs to other subgroups (via supermaster) if the local group is busy. We show in the experimental section that this strategy of using the supermaster route pays off significantly and ensures the system is load balanced.

Master: the primary responsibility of a master is to ensure all consumers in its subgroup are always busy with alignment computation. Given that pairs could take varying time for alignment, it is more desirable to have the local consumers request for pairs from the local master, than have the master push pairs to its local consumers. Furthermore, to prevent work starvation at the consumers, it is important the master responds in a timely fashion to consumer requests. The hierarchical strategy of maintaining small subgroups helps alleviate this to a certain extent. Another challenge for the master is to accommodate the irregular rate at which its local producers are supplying new pairs. A fast supply rate could overrun the local pair buffer. Ideally, we could store as many pairs as can be stored at the fixed size M_{buf} at the master; however, assuming a protocol where the pairs stored on a local master cannot be redistributed to other subgroups, pushing all pairs into a master node may introduce parallel bottlenecks during the ending stages. The above challenges are overcome as

Algorithm 1 Producer

1. Request a batch of subtrees from supermaster
2. **while true do**
3. $T_i \leftarrow$ received subtrees from supermaster
4. **if** $T_i = \emptyset$ **then**
5. break while loop
6. **else**
7. **repeat**
8. **if** P_{buf} is not FULL **then**
9. Generate at most b_1 pairs from T_i
10. Insert new pairs into P_{buf}
11. **end if**
12. **if** $send_{P \to M}$ completed **then**
13. Extract at most b_1 pairs from P_{buf}
14. $send_{P \to M} \leftarrow$ *Isend* extracted pairs to master
15. **end if**
16. **until** $T_i = \emptyset$
17. Request a batch of subtrees from supermaster
18. **end if**
19. **end while**
20. /* Flush remaining pairs */
21. **while** $P_{buf} \neq \emptyset$ **do**
22. Extract at most b_1 pairs from P_{buf}
23. **if** $send_{P \to M}$ completed **then**
24. $send_{P \to M} \leftarrow$ *Isend* extracted pairs to master
25. **end if**
26. **if** $send_{P \to S}$ completed **then**
27. $send_{P \to S} \leftarrow$ *Isend* extracted pairs to supermaster
28. **end if**
29. **end while**
30. Send an END signal to Supermaster

follows (see Algorithm 2).

Initially, to ensure that there is a steady supply and dispatch of pairs, the master listens for messages from both its producers and consumers. However, once $|M_{buf}|$ reaches a preset limit called τ, the master realizes that its suppliers (could be producers or supermaster) have been overactive, and therefore shuts off listening to its suppliers, while only dispatching pairs to its consumers until $|M_{buf}| \leq \tau$. This way, priority is given to consumer response as long as there are sufficient pairs in M_{buf} for distribution, while at the same time, preventing buffer overruns from happening due to an aggressive producer.

On the other hand, when the local set of producers cannot provide pairs in a timely fashion, which could happen at the ending stages when the subtree list has been exhausted, the supermaster could help provide pairs from other subgroups. To allow for this feature, the master opens its listening port to the supermaster as well, whenever it does it to the local producers.

As for serving consumers, the master maintains a priority queue, which keeps track of the states of the work buffers at its consumers based on the latter's most recent status report. The priority represents the criticality of the requests sent from consumers, and is defined based on the number of the pairs left at the consumers' C_{buf}. Accordingly the master dispatches work to the consumers. This implies that the master, instead of pushing pairs on to consumers, waits for consumers to take

the initiative in requesting pairs, while reacting in the order of their current workload status.

While frequent updates from consumers could help the master to better assess the situation on each consumer, such a scheme will also increase communication overhead. As a tradeoff, we implement a priority queue by maintaining three levels of priority depending on the C_{buf} size: $\frac{1}{2}$-empty, $\frac{3}{4}$-empty, and completely empty, in increasing order of priority.

Consumer: the primary responsibility of a consumer is to compute optimal alignments using the Smith-Waterman algorithm [Smith and Waterman, 1981] for the pairs allocated to it by its master and output edges for pairs that succeed the alignment test. One of the main challenges in consumer design to ensure the availability of sequences for which alignment is being performed, as the entire sequence set S cannot be expected to fit in local memory for large inputs. To fetch sequences not available in local memory, we considered two options: one is to use I/O (assuming all consumers have access to a shared file system with the sequence file); and the second option is to fetch them over the network intraconnect from other processors that have them. Intuitively, the strategy of using I/O to fetch unavailable sequences can be expected to incur large latency because the batch of sequences to be aligned at any given time could be arbitrary, thereby implying random I/O calls. Unless there is access to a efficient parallel I/O system, such a strategy is not likely to scale to

Algorithm 2 Master

```
1. τ: predetermined cutoff for the size of M_buf
2. Q: priority queue for consumers
3. while true do
4.    /* Recv messages */
5.    if |M_buf| > τ then
6.       msg ← post Recv for consumers
7.    else
8.       msg ← post open Recv
9.       if msg ≡ pairs then
10.         Insert pairs into M_buf
11.         if msg ≡ END signal from supermaster then
12.            break while loop
13.         end if
14.      else if msg ≡ request from consumer then
15.         Place consumer in the appropriate priority queue
16.      end if
17.   end if
18.   /* Process consumer requests */
19.   while |M_buf| > 0 and |Q| > 0 do
20.      Extract a highest priority consumer, and send appropriate amount of pairs
21.   end while
22. end while
23. /* Flush remaining pairs to consumers */
24. while |M_buf| > 0 do
25.   if |Q| > 0 then
26.      Extract a highest priority consumer, and send appropriate amount of pairs
27.   else
28.      Waiting consumer requests
29.   end if
30. end while
31. Send END signals to all consumers
```

larger system sizes. On the other hand, using the intraconnect network could also potentially introduce network latencies, although the associated magnitude of such latencies can be expected to be much less when compared to I/O latencies in practice. In addition, if implemented carefully network related latencies can be effectively masked out in practice (as will be shown in the experimental results).

To test and compare these two models, we implemented both two versions: $pGraph_{nb}$ that uses nonblocking communication calls and $pGraph_{I/O}$ that uses I/O to do sequence fetches. As a third alternative option one can also think of using MPI one-sided communications (instead of nonblocking calls), particularly since the sequence fetches are read-only operations and therefore it becomes unnecessary to involve the remote processor during fetch. However, with one-sided communications, the problem lies in arranging these calls. Performing a separate one-sided call for every sequence that needs to be fetched at any given time is not a scalable option because that would mean that the number of calls is proportional to the number of pairs aligned in the worst case. On the other hand, aggregating the sequence requests by their source remote processor and issuing a single one-sided call to each such processor runs the disadvantage of fetching more sequence information than necessary. This is because one-sided calls can only fetch in windows of contiguously placed sequences and will therefore bring in unwanted sequences that could be between two required sequences. Due to these constraints, we did not implement a one-sided version. In

what follows, we present the consumer algorithm that uses network for sequence fetching. The details for the I/O version should immediate follow from the description for $pGraph_{nb}$ and are omitted.

The consumer for $pGraph_{nb}$ follows Algorithm 3. Each consumer maintains a fixed size pair buffer C_{buf} and a sequence cache $\mathcal{S}_c$. The sequence cache ($\mathcal{S}_c$) is divided into two parts: (i) a static sequence cache $\mathcal{S}_c^s$ of size $O(\frac{m}{c})$ (preloaded from I/O); and (2) a fixed-size dynamic sequence cache $\mathcal{S}_c^d$ — a transient buffer to store dynamically fetched sequences from other consumers. During initialization, the consumers within *each subgroup* load the input sequence set S into their respective $\mathcal{S}_c^s$ in a distributed manner such that each consumer gets a unique contiguous $O(\frac{m}{c})$ fraction of input bytes. The assumption that the collective memory of all the c consumers in a subgroup is sufficient to load S is without loss of generality because the subgroup size can always be increased to fit the input size if necessary. The characteristic of this application in practice is that thousands of processors are needed to serve the purpose of computation, while the memory on tens of processors are typically sufficient to fit the input sequence data. The strategy of storing the entire sequence set within each subgroup also has the advantage that communications related to sequence fetches can be kept local to a subgroup, thereby reducing hotspot occurrences.

When a consumer receives a batch of new pairs from its master, it first identifies the sequences which are not present in $\mathcal{S}_c^s$ and $\mathcal{S}_c^d$, and subsequently sends out sequence

requests to those consumers in the same subgroup that contains those sequences. When a consumer receives a batch of requests from another consumer, it packs the related sequences and dispatch them using a nonblocking send. When the remote sequences arrive, the receiving consumer unpacks the sequences into $\mathcal{S}_c^d$. A separate counter is maintained with each sequence entry in $\mathcal{S}_c^d$ to keep track of the number of pairs in C_{buf} requiring that sequence at any time. If the counter becomes zero at any stage, then the memory allocated for the sequence is released. The dynamic cache is intended to serve as a virtual window of sequences required in the recent past, and could help reduce the net communication volume. In fact we observed that about 60% savings (as will be shown in the results section). Furthermore, the worst-case dynamic sequence cache size is proportional to $2 \times |C_{buf}|$.

The consumer also sends reports of its C_{buf} size to its local master in a timely fashion. The states are $\frac{1}{2}$-empty, $\frac{3}{4}$-empty, and completely empty. Once a status is sent, the consumer continues to process the remaining pairs in C_{buf}.

If C_{buf} becomes empty, the consumer sends an *empty* message to inform master that it is starving and waits for the master to reply.

Supermaster: the primary responsibility of the supermaster is to ensure that both the pair generation workload and pair alignment workload are balanced across subgroups. To achieve this, the supermaster follows Algorithm 4. At any given iteration, the supermaster is either serving a producer or a master. For managing the

Algorithm 3 Consumer

1. $\Delta = \{0, \frac{1}{4}, \frac{1}{2}\}|C_{buf}|$: **empty, quarter, half** buffer status
2. n_s: number of sequences to be cached statically
3. $\mathcal{S}_c^s$: static sequence cache
4. $\mathcal{S}_c^d$: dynamic sequence cache
5. $recv \leftarrow$post nonblocking receive
6. $\mathcal{S}_c^s \leftarrow$ load n_s sequences from I/O
7. **while true do**
8. **if** $recv$ completed **then**
9. **if** Sequence request from consumer c_k **then**
10. Pack sequences and send them out to c_k
11. $recv \leftarrow$post nonblocking receive
12. **else if** Sequences from other consumer **then**
13. $\mathcal{S}_c^d \leftarrow$ unpack received sequences
14. $recv \leftarrow$post nonblocking receive
15. **else if** Pairs from master **then**
16. Insert pairs into C_{buf}
17. Identify sequences to fetch from others
18. Send sequence requests to other consumers
19. $recv \leftarrow$post nonblocking receive
20. **end if**
21. **else**
22. **if** $|C_{buf}| > 0$ **then**
23. Extract next pair (i, j) from C_{buf}
24. **if** $s_i, s_j \in \mathcal{S}_c^s \cup \mathcal{S}_c^d$ **then**
25. Align sequences s_i and s_j
26. Output edges (s_i, s_j) if they pass cutoffs
27. **else**
28. Append pair (i, j) at the end of the C_{buf}
29. **end if**
30. **if** $|C_{buf}| \in \Delta$ **then**
31. Report $|C_{buf}|$ status to master
32. **end if**
33. **end if**
34. **end if**
35. **end while**

pair generation workload, the supermaster assumes the responsibility of distributing subtrees (in fixed size batches) to individual producers. The supermaster, instead of pushing subtree batches to producers, waits for producers to request for the next batch. This approach guarantees that the run-time of the producers (and not necessarily the number of subtrees processed) is balanced at program completion.

The second task of the supermaster is to serve as a conduit for pairs to be redistributed across subgroup boundaries. To achieve this, the supermaster maintains a local buffer, S_{buf}. Producers can choose to send pairs to supermaster if their respective subgroups are saturated with alignment work. The supermaster then decides to redirect the pairs (in batches of size b_1) to masters of other subgroups, depending on their respective response rate (dictated by their current workload). This functionality is expected to be brought into effect at the ending stages of producers' pair generation, when there could be a few producers that are still churning out pairs in numbers while other producers have completed generating pairs. As a further step toward ensuring load balanced distribution at the producers' ending stages, the supermaster sends out batches of a reduced size, $\frac{b_1}{2}$, in order to compensate for the deficiency in pair supply. Correspondingly, the masters also reduce their batch sizes proportionately at this stage. As shown in the experimental section, the supermaster plays a key role in load balancing of the entire system.

Algorithm 4 Supermaster

1. Let $P = \{p_1, p_2, ...\}$ be the set of active producers
2. $recv_{S \leftarrow P} \leftarrow$ Post a nonblocking receive for producers
3. **while** $|P| \neq 0$ **do**
4. /* Serve the masters*/
5. **if** $|S_{buf}| > 0$ **then**
6. $m_i \leftarrow$ Select master for pairs allocation
7. Extract and *Isend* b_1 pairs to m_i
8. **end if**
9. /* Serve the producers*/
10. **if** $recv_{S \leftarrow P}$ completed **then**
11. **if** $msg \equiv$ subtree request **then**
12. Send a batch of subtrees (T_i) to corresponding producer
13. **else if** $msg \equiv$ pairs **then**
14. Insert pairs in S_{buf}
15. **end if**
16. $recv_{S \leftarrow P} \leftarrow$ Post a nonblocking receive for producers
17. **end if**
18. **end while**
19. Distribute remaining pairs to all masters in a round-robin way
20. Send END signals to all masters

3.3.3 Redundant sequence removal

In microbial environments, some organisms are more abundant than others. To interpret this abundance in sequence level, some sequences are expected to be over represented. These over-represented (*"redundant"*) sequences do not provide any additional information during homology detection or for clustering. So it is better to eliminate them before the subsequent analysis. For a sequence to be treated as "redundant" regarding to another sequence, the criteria are defined as: (1) at least 98% of matched regions should be included in the alignment; (2) 95% of the shorter sequence should participate in the alignment; and (3) alignment score is greater than 95% of the optimal self-score of the shorter sequence. If two sequences, say s_1 and s_2 satisfy this criteria, we denoted it as $\tau(s_1, s_2) = 1$; otherwise $\tau(s_1, s_2) = 0$. Based on this redundant criteria, the redundant sequence problem can be formally defined as follows:

Problem 2 *Given a set* $S = \{s_1, s_2, \cdots, s_n\}$ *of* n *sequences and a predefined redundant criteria* τ*, the redundancy removal problem is to find a subset* S' *of* S*, such that* $\forall s_i, s_j \in S', i \neq j$*,* $\tau(s_i, s_j) = 0$*.*

Ideally we would expect to remove these redundant sequences before the sequence homology detection (graph construction) step; however to identify these re-

dundant sequences, we have to perform a similar operation like "graph construction" only with more stringent criteria. This identification process will be as time-consuming as the graph construction itself. A careful observation will find out that if one sequence is treated as redundant comparing to another, then an "edge" must also be preserved between the two sequences. In that case, we could integrate the redundant removal step into the graph construction process. The basic idea is as follows: if two sequences pass the predefined "edge" criteria, then it is more likely for them to meet the redundant criteria (the "edge" criteria is a necessary but not sufficient condition to confirm a sequence as redundant); if not, no redundancy check is required at all. If two sequences pass the predefined redundancy cutoffs, then in our case the shorter sequence is marked down as redundant. After the graph construction process, all the marked sequences are removed thereafter for subsequent analysis.

As indicated in graph construction part, each sequence is directed mapped into a vertex in graph $G = (V, E)$. In order to remove these redundant sequences (vertices in G), we basically enumerate all the constructed edges in graph G, and if either vertex of the edges overlap with the marked vertices, then the edge and associated vertices are eliminated from the graph G.

3.4 Experiments

3.4.1 Implementation

The *pGraph* code was implemented in C/MPI, and the "redundant sequence removal" step was implemented as post-processing in Perl. the All parameters described in the algorithm section were set to values based on preliminary empirical tests. The default settings are as follows: b_1 =30,000; b_2 =2,000; $|P_{buf}| = 1 \times 10^7$; $|M_{buf}| = 6 \times 10^4$; $|C_{buf}| = 6 \times 10^3$; $|S_{buf}| = 4 \times 10^6$. Two sequences are said to be "homologous", if they share a local alignment with a minimum 40% identity and if the alignment covers at least 80% of the longer sequence.

The software and related documentation is freely available as open source and can be obtained through link: `http://code.google.com/p/psgraph/`.

3.4.2 Experimental setup

Input data: The *pGraph* implementations were tested using an arbitrary collection of 2.56×10^6 (n) amino acid sequences representing an ocean metagenomic data set available at the CAMERA metagenomics data archive [CAMERA, 2011]. The sum of the length of the sequences (m) in this set is 390,345,218, and the mean$\pm\sigma$ is 152.48 ± 167.25; the smallest sequence has 1 residue and longest 32,794 residues.

Smaller size subsets containing 20K, 40K, 80K, ..., 1.28×10^6 were derived and used for scalability tests.

Experimental platform: All tests were performed on the *Chinook* supercomputer at the EMSL facility in Pacific Northwest National Laboratory. This is a 160 TF supercomputer running Red Hat Linux and consists of 2,310 HP DL185 nodes with dual socket, 64-bit, Quad-core AMD 2.2 GHz Opteron processors with an upper limit of 4 GB RAM per core. The network interconnect is Infiniband. A global 297 TB distributed LUSTRE file system is available to all nodes.

pGraph-**specific settings:** Even though 4 GB RAM is available at each core, for all runs we set a strict memory upper limit for usage to $O(\frac{m}{c})$ per MPI process, where c is the number of consumers in a subgroup. This was done to emulate a generic use-case on any distributed memory machine including those with limited memory per core. At the start of execution, all consumers in a subgroup load the input sequences in a distributed even fashion such that each consumer receives a unique $O(\frac{m}{c})$ fraction of the input. The locally available set of sequences is referred to as the "static sequence cache". Any additional sequence that is temporarily fetched into local memory during alignments is treated as part of a fixed size "dynamic sequence cache".

To generate the suffix tree index required for all input sets, a construction code from one of our earlier developments [Kalyanaraman et al., 2007] was used. The suffix tree index for each input is generated as a forest of subtrees, one for

each unique $k-mer$ in the input. We used $k = 4$ for all trees. The tree index statistics for the different input sets are shown in Table 3.1. A single CPU was used to generate the trees for all our experiments because the tree construction is quick and expected to scale linearly with input size, as shown in the table. For larger inputs, any of the already available parallel implementations can be used [Ghoting and Makarychev., 2009, Kalyanaraman et al., 2007]. Table 3.1 also shows the number of subtrees generated for each input set. As k was used, the total for all our runs, we assume that the tree index is already built using any method of choice and stored in the disk.

For all the performance results presented in Section 3.4.4, we set the subgroup size to 16 and the number of producers per subgroup to 2 (to approximate a producer:consumer ratio of 1:7 within each subgroup). The effect of changing these parameters are later studied in Section 3.4.3.

3.4.3 Parametric studies

We studied the effect of subgroup size on $pGraph_{nb}$'s performance by varying the subgroup sizes from 8, 16, 32, ... to 512, and keeping the total processor size fixed at 1,024 on the 640K input. In all our experiments, a producer:consumer ratio

No. input sequences	Total sequence length	No. subtrees in the forest	No. tree nodes	Construction time (in secs; single CPU)
20K	3,852,622	133,639	5,721,111	3
40K	8,251,063	149,501	12,318,567	6
80K	20,600,384	158,207	30,952,989	26
160K	43,480,130	159,596	66,272,332	56
320K	86,281,743	159,991	128,766,176	108
640K	160,393,750	160,000	237,865,379	205
1,280K	222,785,671	160,000	306,132,294	300
2,560K	392,905,218	160,000	533,746,500	520

Table 3.1: Sequence and suffix tree index statistics for different input sets.

of 1:7 ratio was approximately maintained within each subgroup to reflect the average pair generation to alignment cost ratio. For example, a subgroup with 8 processors will contain 1 producer, 1 master and 6 consumers; whereas a subgroup with 512 processors will contain 64 producers, 1 master and 447 consumers. Note that a larger group size implies less number of subgroups to manage for the supermaster and also more importantly, more number of consumers to contribute to alignment computation. However, as the number of consumers per subgroup increase, the overheads associated with the local master response time and for sequence fetches from other consumers also increase. Therefore, it is increasingly possible that a consumer spends more time waiting (or idle) for data. Figure 3.2 shows the parallel run-time and the portion of it that an average consumer spends idle waiting either for pairs from the

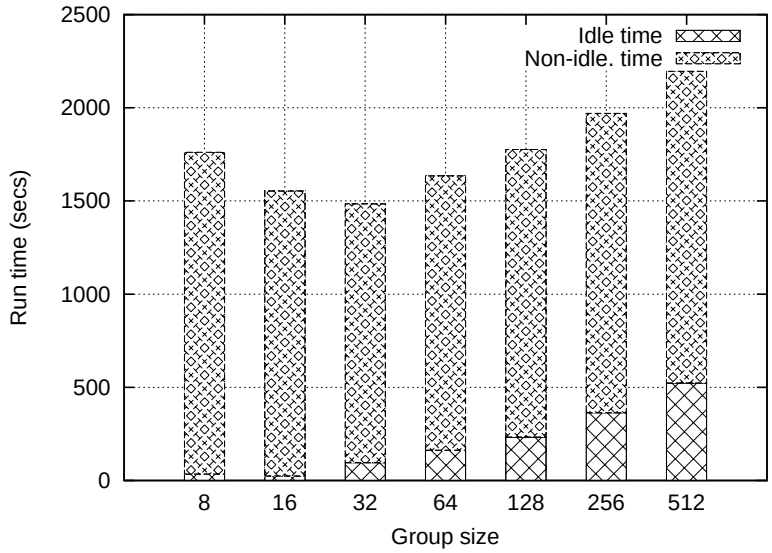

Figure 3.2: Chart showing the effect of changing the group size on performance. All runs were performed on the 640K input, keeping the total number of processors fixed at 1,024.

local master or for sequences from other consumers. As expected, we find that the total time reduces initially due to faster alignment computation, before starting to increase again due to increased consumer idle time. The figure also shows an empirically optimal run-time is achieved when the subgroup size is between 16 and 32. Even though this optimal breakeven point is data dependent, the general trend should hold for other inputs as well.

3.4.4 Performance evaluation

Comparative evaluation: $pGraph_{I/O}$ vs. $pGraph_{nb}$: at first, we compare the two versions of our software, $pGraph_{I/O}$ and $pGraph_{nb}$, which use I/O and non-blocking communication, respectively, for fetching sequences not in either of the local sequence caches during alignment at consumers. Figure 3.3 shows the runtime breakdown of an average consumer under each implementation, on varying number of processors for the 640K input. Both implementations scale linearly with increasing processor size. However, in $pGraph_{I/O}$, alignment time accounted only for $\sim 80\%$ of the total run-time, and the remaining 20% of the time is dominated primarily by I/O, for all processor sizes. In contrast, for $pGraph_{nb}$ nearly all of the run-time was spent performing alignments leaving the overhead associated with non-blocking communication negligible. Notably, the non-blocking version is 20% faster than the I/O version. The trends observed hold for other data sets tested as well (data not shown). The results show the effectiveness of the masking strategies used in the non-blocking implementation and more importantly, its ability to effectively eliminate overheads associated with dynamic sequence fetches through the network. This coupled with the linear scaling behavior observed for $pGraph_{nb}$ makes it the implementation of choice.

Note that the linear scaling behavior of $pGraph_{I/O}$ can be primarily attributed to the availability of a fast, parallel I/O system such as Lustre. Such scaling cannot

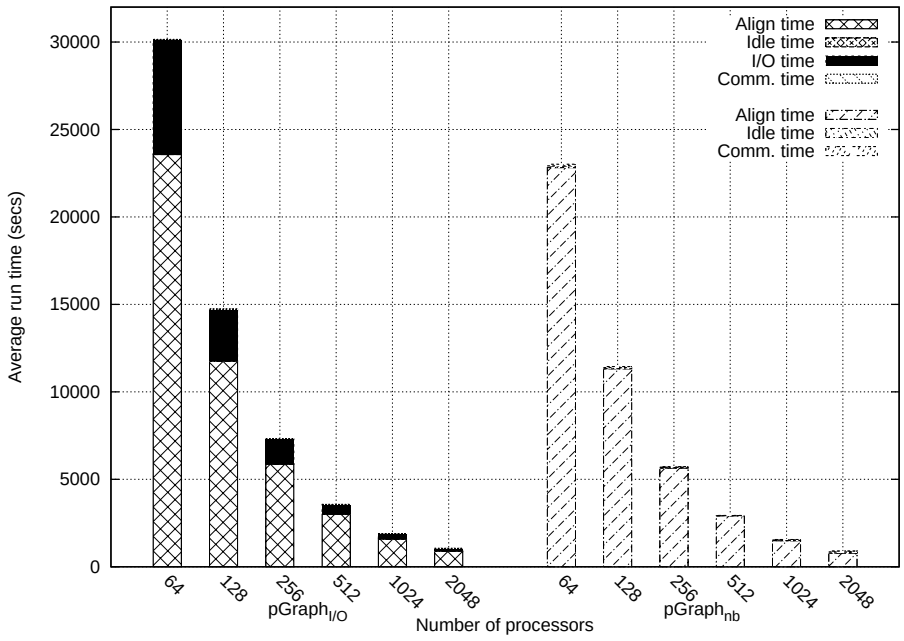

Figure 3.3: Comparison of the I/O and non-blocking communication versions of *pGraph*. Shown are the runtime breakdown for an average consumer between the two versions. All runs were performed on the $640K$ input sequence set. The results show the effectiveness of the non-blocking communication version in eliminating sequence fetch overhead.

be expected for systems that do not have a parallel I/O system in place. In what follows, we present all of our performance evaluation using only $pGraph_{nb}$ as our default implementation.

Performance evaluation for $pGraph_{nb}$:

Input	Number of processors (p)								#pairs
#seqs. (n)	16	32	64	128	256	512	1,024	2048	(in millions)
20K	398	192	94	49	26	14	9	-	6.5
40K	1,217	583	286	143	73	37	20	-	16.9
80K	19,421	9,260	4,481	2,243	1,146	616	373	-	48.5
160K	-	-	7,666	3,837	1,978	1,011	574	356	125.6
320K	-	-	16,283	8,056	4,061	2,082	1,060	623	365.7
640K	-	-	23,102	11,481	5,739	2,942	1,561	893	590.1
1,280K	-	-	-	32,113	16,042	8,014	4,031	2,066	2,410.4
2,560K	-	-	-	124,884	62,222	31,103	15,639	7,975	5,258.3

Table 3.2: The run-time (in seconds) for $pGraph_{nb}$ on various input and processor sizes. An entry '-' means that the corresponding run was not performed. The last column shows the number of pairs aligned (in millions) for each input as a measure of work.

Table 3.2 shows the total parallel runtime for a range of input sizes (20K ... 2,560K) and processor sizes (16 ... 2,048). The large input sizes scale linearly up to 2,048 processors and more notably, inputs even as small as 20K scale linearly up to 512 processors. The speedup chart is shown in Figure 3.4. All speedups are calculated relative to the least processor size run corresponding to that input. The smallest run had 16 processors because it is the subgroup size. The highest speedup ($2,004\times$) was achieved for the 2,560K data on 2,048 processors. Figure 3.5 shows the parallel efficiency of the system. As shown, the system is able to maintain an efficiency above 90% for most inputs. Also note that for several inputs, parallel efficiency slightly *increases* with processor size for smaller number of processors (e.g., 80K on $p : 32 \rightarrow$

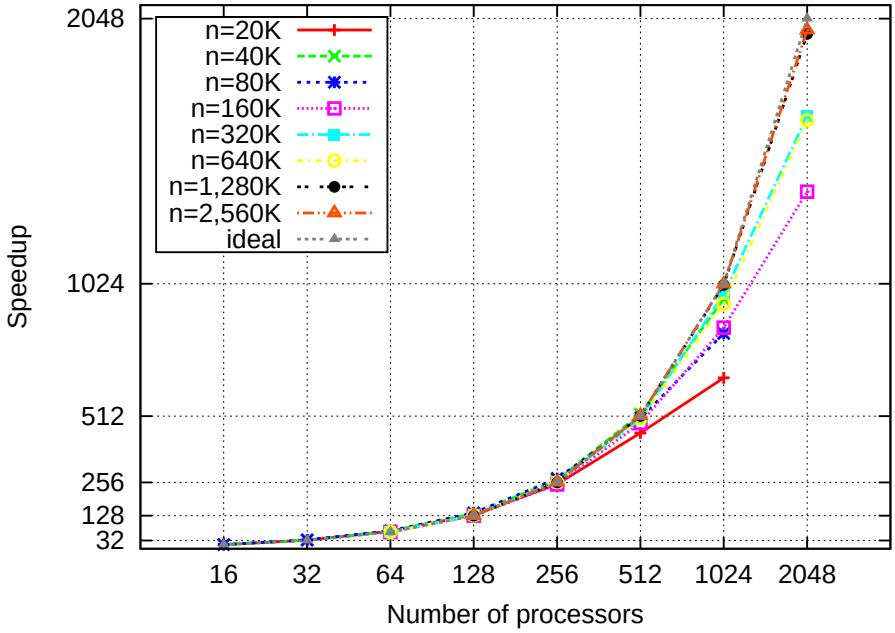

Figure 3.4: Speedup of *pGraph*. The speedup computed are relative, and because the code was not run on smaller processor sizes for larger inputs, the reference speedups at the beginning processor size were assumed at linear rate — e.g., a relative speedup of 64 was assumed for 160K on 64 processors. This assumption is consistent with the linear speedup trends observed at that processor size for smaller inputs.

64). This super-linear behavior can be attributed to the minor increase in the number of consumers (relative to the whole system size) — i.e., owing to the way in which the processor space is partitioned, the number of consumers more than doubles when the whole system size is doubled (e.g., when p increases from 16 to 32, the number of

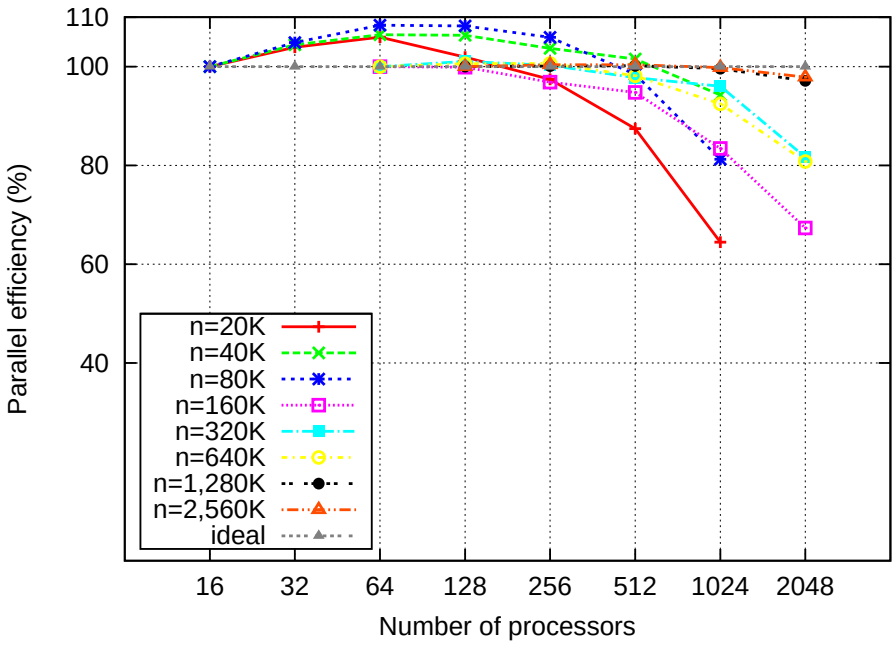

Figure 3.5: Parallel efficiency of *pGraph*. The parallel efficiency are also relatively computed based on the smallest run on that dataset.

consumers increases from 12 to 25). And this increased availability contributes more significantly for smaller system sizes — e.g., when p increases from 16 to 32, the one extra consumer adds 4% more consumer power to the system. The effect however diminishes for larger system sizes.

Table 3.2 also shows run-time increase as a function of input number of sequences. Although this function cannot be analytically determined because of its input-dependency, the number of alignments needed to be performed can serve as a

good indicator. However, Table 3.2 shows that in some cases the run-time increase is not necessarily proportional to the number of pairs aligned — e.g., note that a 3× increase in alignment load results in as much as a 16× increase in run-time, when n increases from 40K to 80K. Upon further investigation, we found the cause to be the difference in the sequence lengths between both these data sets — both mean and standard deviation of the sequence lengths increased from 205±118 for the 40K input to 256±273 for the 80K input, thereby implying an increased cost for computing an average unit of alignment.

To better understand the overall system's linear scaling behavior and identify potential improvements, we conducted a thorough system-wide study. All runs were performing using $n = 640K$ as the case study.

Consumer behavior: At any given point of time, a consumer in $pGraph_{nb}$ is in one of the following states: i) *(align)* compute sequence alignment; or ii) *(comm)* communicate to fetch sequences or serve other consumers, or send pair request to master; or iii) *(idle)* wait for master to allocate pairs. As shown in Figure 3.3, an average consumer in $pGraph_{nb}$ spends well over 98% of the total time computing alignments. This desired behavior can be attributed to the combined effectiveness of our masking strategies, communication protocols and the local sequence cache management strategy. The fact that the idle time is negligible demonstrates the merits of sending timely requests to the master depending on the state of the local pair buffer.

Despite the fact that sequence requests are random and are done asynchronously, the contribution due to communication is negligible both at the senders and receivers. Keeping a small subgroup size (16 in our experiments) is also a notable contributor to the reason why the overhead due to sequence fetches is negligible. For larger subgroup sizes, this asynchronous wait times can increase (see Section 3.4.3).

The local sequence management strategy also plays an important role. Note that each consumer only stores $O(\frac{m}{c})$ characters of the input in the static cache. Figure 3.6 shows the statistics relating to sequence fetches carried out at every step as the algorithm proceeds at an arbitrarily chosen consumer. As the top chart shows, the probability of finding a sequence in the local static cache is generally low, thereby implying that most of the sequences required for alignment computation needed to be fetched over network. While the middle chart confirms this high volume of communication, it can be noted that the peaks and valleys in this chart do not necessarily correspond to that of the top chart. This is because of the temporary availability of sequences in the fixed size dynamic sequence cache (bottom chart), which serves to reduce the overall number of sequences fetched from other consumers by about 60%.

Master behavior: The master within any subgroup is in one of the following states at any given point of execution: i) *(idle)* waiting for consumer requests or new pairs from the local producer(s) or the supermaster; or ii) *(comm)* sending pairs to a consumer; or iii) *(comp)* performing local operations to manage subgroup. Figure 3.7

shows that the master is available (i.e., idle) to serve its local subgroup nearly all of its time. This shows the merit of maintaining small subgroups in our design. The effectiveness of the master to provide pairs in a timely fashion to its consumers is also important. Figure 3.8 shows the status of a master's pair buffer during the course of the program's execution. As can be seen, the master is able to maintain the size of its pair buffer steadily despite the non-uniformity between the rates at which the pairs are generated at producers and processed in consumers. The sawtooth pattern is because of the master's receiving protocol which is to listen to only its consumers when the buffer size exceeds a fixed threshold.

Producer behavior: The primary responsibility of producers is to keep the system saturated with work by generating sequence pairs from trees and sending them to the local master (or the supermaster) in fixed size batches. Figure 3.9 shows the run-time and number of pairs generated at each producer. As can be observed, there is considerable variability in the number of pairs supplied by each producer, although all producers finish roughly at the same time. This confirms the irregular behavior of the pair generation phase, which is a result of the irregular overheads associated with tree processing. The results also shows the effectiveness of the dynamic tree distribution strategy deployed by the supermaster.

Note that, even with two producers per subgroup, the pair generation time for all producers is $\sim$400s, which is roughly about 25% of the total execution time for the

640K input. For larger data sets, pair generation could consume a substantial part of the run-time and therefore keeping the roles of the master and producers separate is essential for scalability. Also, the increased memory capacity through using multiple producers to stock pairs that are pending alignment computation further supports a decoupled design.

Supermaster behavior: At any given point of time, the system's supermaster is in one of the following states: i) *(producer polling)* checking for messages from producers, to either receive tree request or pairs for redistribution; ii) *(master polling)* checking status of masters to redistribute pairs. Figure 3.10 shows that the supermaster spends roughly about 25% of its time the polling the producers and the remainder of the time polling the masters. This is consistent with our empirical observations, as producers finish roughly in the first 25% of the program's execution time, and the remainder is spent on simply distributing and computing the alignment workload.

Does the supermaster's role of redistributing pairs for alignment across subgroups help? To answer this question, we implemented a modified version — one that uses supermaster only for distributing trees to producers but *not* for redistributing pairs generated across groups. This modified implementation was compared against the default implementation, and the results are shown in Figure 3.11. As is evident, the scheme without pair redistribution creates skewed run-times across subgroups and introduces bottleneck subgroups that slow down the system by up to 40%. This

is expected because a subgroup without support for redistributing its pairs may get overloaded with more pairs and/or pairs that need more alignment time, and this combined variability could easily generate nonuniform workload. This shows that the supermaster is a necessary intermediary among subgroups for maintaining overall balance in both pair generation and alignment.

3.5 Conclusion

In this chapter, we presented a novel parallel algorithm and implementation to efficiently parallelize the construction of sequence homology graphs on large-scale protein sequence data sets using distributed memory computers. Coarse-level parallelism for this problem has been lacking in practice. The proposed parallel design is a hybrid of multiple-master/worker and producer-consumer models, which effectively addresses the unique set of irregular computation issues and input data availability issues. The new implementation demonstrates linear scaling on up to 2,048 processors that were tested, for a wide range of input sets tested up to 2.56×10^6 metagenomic amino acid sequences. A thorough system-wide study by its components further confirms that the trends observed are likely to hold for larger data sets and for larger processor sizes. A key significance of our new implementation is that it enables users to evaluate large collections of protein sequences using the highly sensitive alignment

computation algorithms.

To put these results in perspective, consider the following comparison with the ocean metagenomics results [Yooseph et al., 2007], which is the largest exercise in protein sequence homology detection to date. The $pGraph_{nb}$ implementation took 7,795 s on 2,048 processors for analyzing a 2.56×10^6 sequence subset of the ocean data set. Based on this, even assuming an absolute worst-case of quadratic explosion of work to 28.6×10^6, we conservatively estimate that $pGraph_{nb}$ would take 566,260 CPU hours. Compare this to the 10^6 CPU hours consumed in [Yooseph et al., 2007] despite the use of the faster albeit less-sensitive BLAST heuristic for evaluating homology.

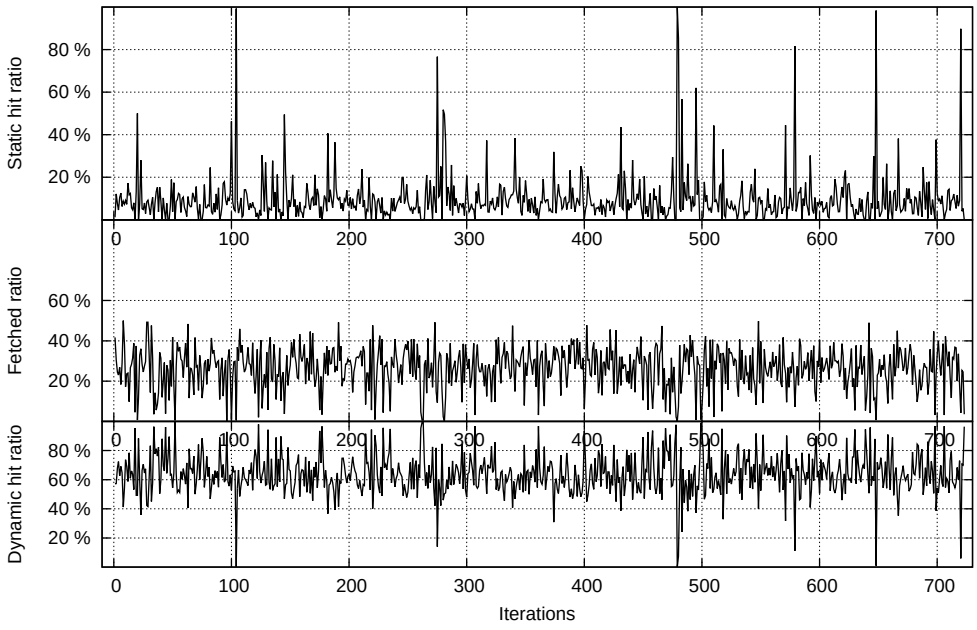

Figure 3.6: Statistics of sequence use (and fetch) on an average consumer ($n = 640K$, $p = 1,024$). The topmost chart shows the percentage of sequences successfully found locally in the static cache during any iteration. The next two charts show the corresponding percentages of sequences that needed to be fetched (communicated) from other consumers, and found locally in the dynamic cache, respectively.

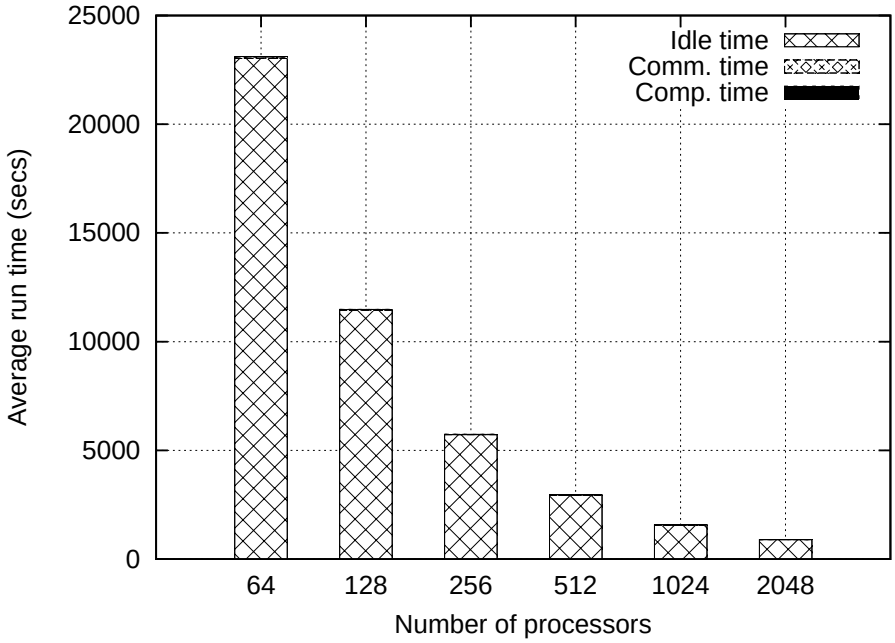

Figure 3.7: Run-time breakdown for an average master ($n = 640K$).

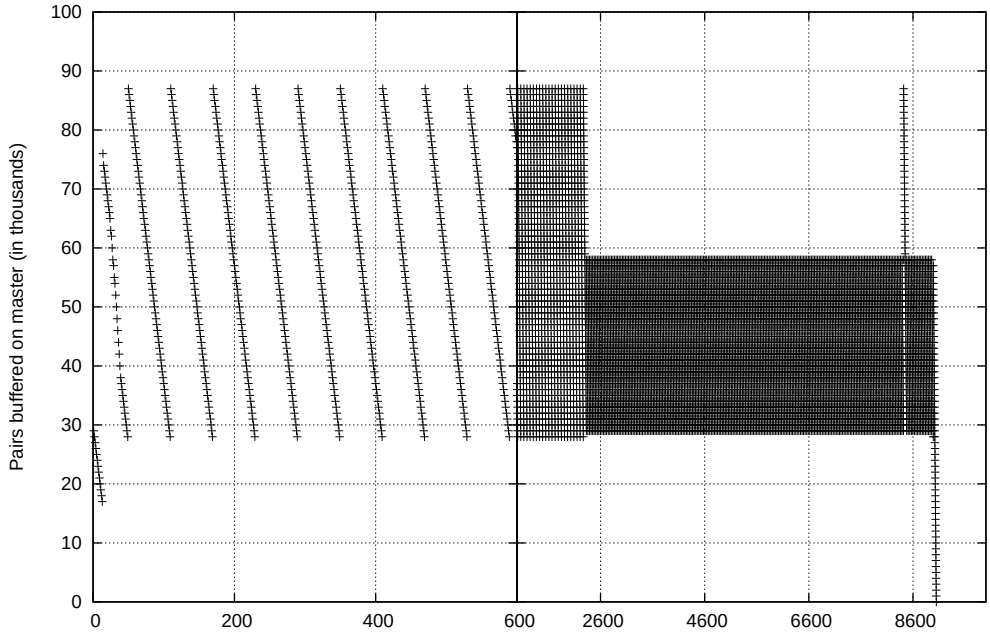

Figure 3.8: The status of M_{buf} on a typical master as execution progresses (subgroup size 16).

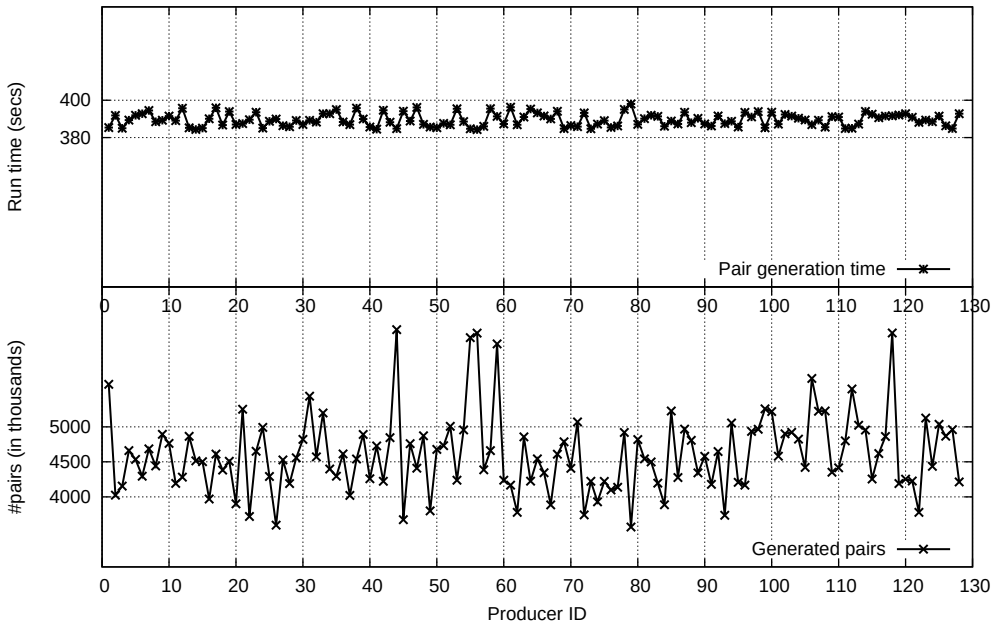

Figure 3.9: Plots showing producer statistics on the number of trees processed, the number of pairs generated and the run-time of each of the 128 producers (i.e., 64 subgroups) for the 640K input.

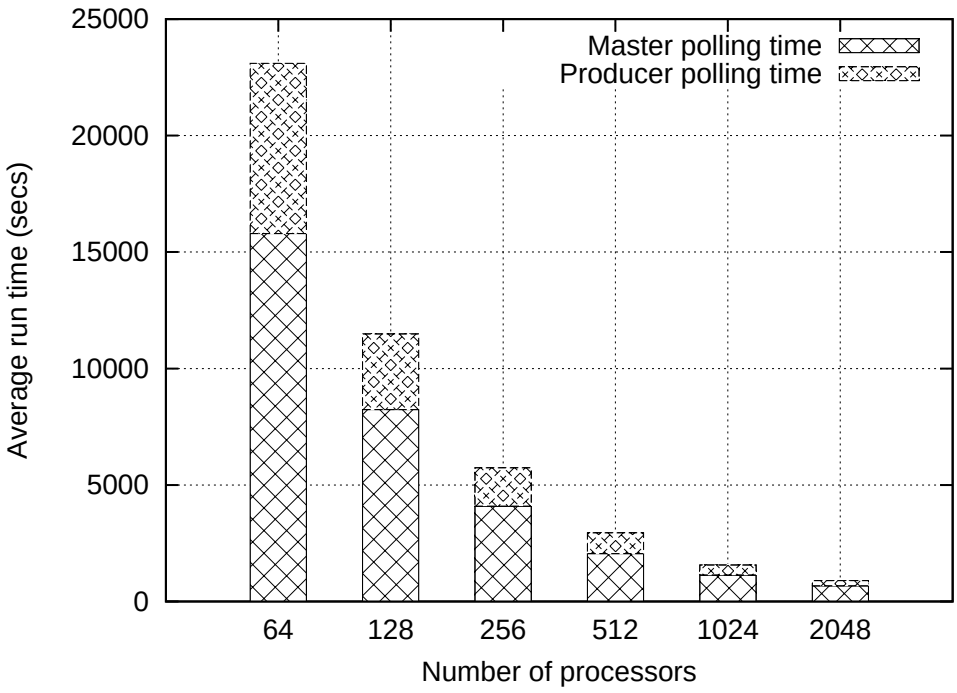

Figure 3.10: Run-time breakdown for the supermaster ($n = 640K$).

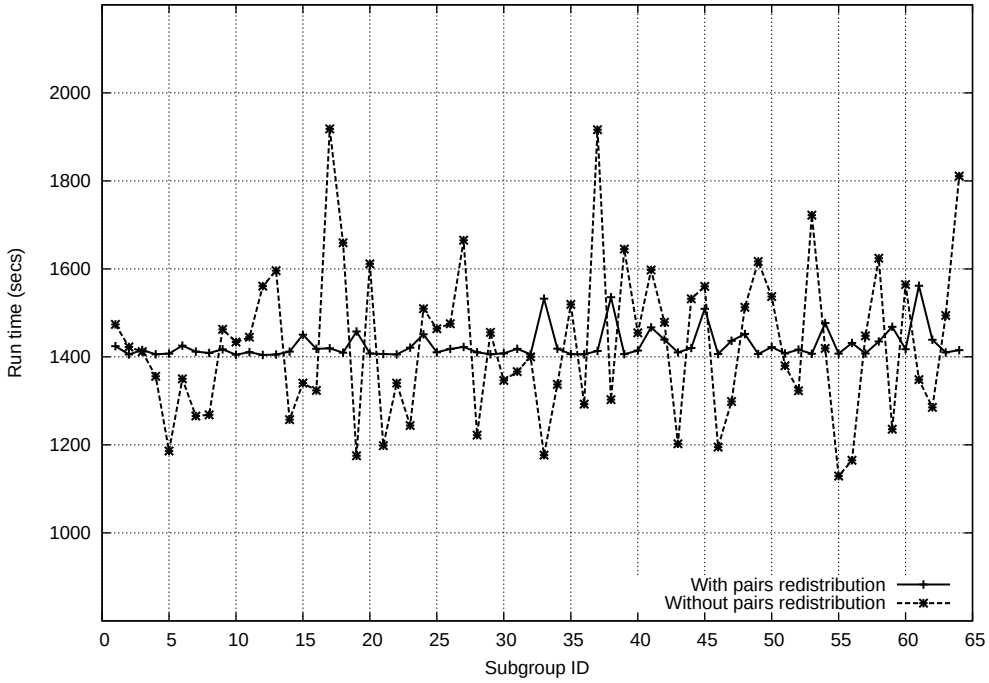

Figure 3.11: The distribution of run-time over 64 subgroups (i.e., $p = 1,024$) for the 640K input, with and without the supermaster's role in pair redistribution. The chart demonstrates that the merits of the supermaster's intervention.

CHAPTER 4. SEQUENCE CLUSTERING

I was gratified to be able to answer promptly. I said I don't know.

— Mark Twain

4.1 Introduction

After constructing the sequence homology graph, it is nature to ask which group of sequences are closely related? In order to answer this question, we need to develop a grouping functionality that clusters sequences together such that in each group every sequence is connected to most of the other sequences in the same group, and the connections among sequences in different groups are sparse. As the sequences in the same group are supposed to be evolutionarily- (or functional-) related, thus functions of a sequence can be deduced from the functions of the other sequences in the same group. For the protein sequences we are addressing here, if two protein sequences have a high sequence similarity, then their structure and functions are very close to each other. So if we could cluster protein sequences based on their sequence similarity, then we could easily deduce the functions for the sequences in the same clusters. More specially, we have a set of known protein sequences (from public repositories)

and a set of unknown protein sequences (from GOS project); after clustering these sequences together, two types of inferences can be made. If a newly derived sequence can be mapped to a group of known sequences (known protein family), then its functional role can be determined based on the known protein family. Otherwise, group of unknown sequences that share similarity with one another but not with any sequences in known protein families can lead to the inference of *de novo* protein families. Either way, the resulting inferences significantly enrich our understanding of the protein universe represented in the underly metagenomic communities.

In graph theoretic domain, the sequence clustering problem can be mapped to *dense subgraph* detection problem. A *dense subgraph* is an arbitrary sized induced subgraph such that "most" pair of the vertices in the subgraph are connected by edges in the subgraph. If there are "fewer" edges connecting the vertices in the subgraph, then the subgraph is called *sparse subgraph*. In extreme case, the dense subgraph is a *clique* which is defined as a graph such that each pair of its vertices are connected. Given the above, the *"dense subgraph detection problem"* can be defined as follows:

Definition 1 *Given graph $G = (V, E)$, dense subgraph detection problem is defined as finding a set of disjoint subsets of V, such that each subset represents a maximal dense subgraph in G.*

In this chapter, we present a heuristic clustering approach, called *pClust* to solve

this challenging problem under the context of the protein family identification. There are two ways to group sequences together: one is to group them into disjoint subsets; and the other is to allow overlaps between subsets (e.g. multi-domain sequences). In this chapter we are primarily focusing on disjoint sequence clustering problem. In order to tackle this sequence clustering problem in large-scale, couple of computational challenges need to be addressed:

1. **Space:** A sequence homology graph of millions of sequences can occupy tens of gigabytes, thereby making it impractical to assume that they will fit in local memory. Whereas, most current graph clustering methods [] assume the input graph fits in the local memory.

2. **Time:** Even if the graph can fit into the memory, how to solve this clustering problem efficiently is another challenge. As we noted above, quadratic efficient algorithms would be prohibitive slow when it comes to millions of sequences in our case.

In what follows, Section 4.2 presents the current state of art for sequence clustering. Section 4.3 presents the heuristic algorithms used to solve this problem. Experimental results are presented in Section 4.4. Finally, Section 4.5 concludes this chapter with future works.

4.2 Related work

Finding a subgraph with the maximum average degree (densest subgraph) can be solved polynomially using flow techniques [Lawler, 2001]; the fastest algorithm used to detect densest subgraph runs in time $O(|E| \cdot |V| \log(|V|^2/|E|))$ [Gallo et al., 1989], which is prohibitive slow in practice given millions of vertices as input. Another related problem is to find the k-densest subgraph: find a subgraph with k vertices such that the average degree of the subgraph induced by the k vertices is maximum. Different from densest subgraph problem, the k-densest subgraph problem is a NP-hard problem (reduction from *clique* problem). Although finding the densest subgraph problem can be solved in polynomial time, finding all dense subgraphs can be achieved through applying the densest subgraph (DS) detection algorithm iteratively. However the required space and time complexity restrict its application in practice.

To speed up this clustering process, several other approximation algorithms [Apweiler et al., 2004, Bateman et al., 2004, Enright et al., 2002, Kriventseva et al., 2001, Olman et al., 2007] are proposed. Although these proposed algorithms are targeting at the arbitrarily sized input graph, they typically can only identify relatively small subgraphs. Another assumption of these algorithms is that the input graph are relatively small and can all fit into the memory, which is not the case in our application. As more and more sequences are produced on a daily basis, the large data size of

the sequences are posing more challenges on the development of sequence clustering algorithms. It is imperative to design a space and time efficient approach to solve this challenging sequence clustering problem.

4.3 Algorithms

Notation: Let $G = (V, E)$ denote the constructed sequence homology graph. Let $CC = \{C_1, C_2, \ldots, C_t\}$ be the set of all the connected components presented in G. Let $\Gamma(v_i) = \{u|(v_i, u) \in E\}$ denote all the vertices which are connected to v_i; this set is also called the *outlinks* or *neighbors* of vertex v_i. $d(v)$ is used to denote the degree of vertex v, and it is defined as the number of vertices adjacent to v.

4.3.1 Connected Components Detection

For a dense subgraph, each vertex contained inside is expected to be connected to most of the other vertices in the same dense subgraph. Therefore, there should be at least a path between any two vertices in a detected dense subgraph. It also implies that *all dense subgraphs must occur within a connected component.*

Observation 1 *Given a graph G, no dense subgraph in G can span across multiple connected components in G.*

Following this observation, the problem of finding dense subgraphs can be reduced to the problem of first enumerating all connected components and then searching each connected component independently for dense subgraphs. In this way, the large problem instance could be broken down into subproblems of much smaller size, and space challenge could be eliminated thereafter. Theoretically, there could be a worst case that all sequences belong to the same connected component, while in practice this possibility was ruled out. Before getting into the details of the dense subgraph detection algorithm, we first focusing on how to detect connected components in a given large graph.

Definition 2 *A connected component C_i of graph G is defined as a maximal subgraph where each vertex has at least one path to every other vetices within the same subgraph.*

The primary goal of this task is to report all connected components $CC = \{C_1, C_2, \ldots, C_t\}$ from previously constructed homology graph $G = (V, E)$. This task can be directly achieved through an union-find (disjoint-set) data structure [Tarjan, 1975]. Initially, all vertices $v \in V$ forms an individual connected component by themselves. If there is an edge connecting any two vertices in two different connected components, then the connected components are collapsed together into one connected component. This collapsing step continues until all the edges in graph G are enumerated. Finally, the resultant connected components are reported as CC. Moreover, to

facilitate the subsequent "dense subgraph" detection step, the edges constructed in graph G are also associated with the vertices in each reported connected components. A detailed description of the algorithm is shown in Algorithm 5.

Algorithm 5 Connect component detection ($G(V, E)$)

1. let $\zeta = \{v | v \in V\}$ be the initial set of connected components.
2. /* find connected components */
3. **for all** $e = (i, j) \in E$ **do**
4. **if** $find(i) \neq find(j)$ **then**
5. $union(i, j)$
6. **end if**
7. **end for**
8. $\zeta \leftarrow \{C_1, C_2, \ldots, C_t\}$
9. /* reconstruct the edges in C_i */
10. **for all** $C_i \in CC$ **do**
11. **for all** $e = (i, j) \in E$ **do**
12. **if** $i \in V(C_i)$ **and** $j \in V(C_i)$ **then**
13. $E(C_i) = \bigcup\{(i, j)\}$
14. **end if**
15. **end for**
16. **end for**
17. $\zeta \leftarrow \{C_1, C_2, \ldots, C_t\}$

As shown in the algorithm, this union-find based connected components algorithm runs in $O(|E|)$, and the space complexity of this algorithm is $O(|V|)$. The linear space and time complexity make this algorithm an ideal candidate for large-scale data input. As only vertices are needed to be maintained in memory, and the edges can be enumerated in a streamed way. The linear feature of this algorithm makes it easily to scale to billions of input sequences.

4.3.2 k-core decomposition

As small clusters (size < 2) are typically of no practice, the input graph can be pre-pruned to remove the vertices which can only fall into the small dense subgraphs. In other words, if a vertex has a degree less than our predefined cutoff, say k, then it cannot fall into any dense subgraph of size at least k. Basically our goal is to prune the input graph such that the degrees of all remaining vertices are greater or equal to k. In this way, some unnecessary work could be eliminated to speed up the subsequent processing. However, note that the removal of a low degree vertex could introduce new low degree to other vertices. Formally, this problem is called k-core [Bollobas, 1984] decomposition problem.

Definition 3 *The k-core of a graph $G = (V, E)$ is defined as an induced subgraph $G' = (V', E')$ such that the degree ($d(v)$) of every vertex in V' is greater or equal to k.*

To find the k-core of a graph, one intuitive way is to remove all vertices with degree less than k in a recursive way. However one challenge is that graph is dynamically changing during the pruning process. A vertex with degree greater than k could become one with degree less than k after pruning of the other vertices, and the graph-pruning stage are dynamically changing the graph. In order to remove the

vertices with degree less than k, we implement a binary-heap based algorithm. Basically all the vertices are stored in a binary min-heap based on their degrees. In this way, the root always maintains the vertex with the minimal degree. If at some point the vertex contained at the root node has a degree greater or equal to k, then the pruning process could be terminated. After removing a vertex in a root node, degree adjustments are required for all the vertices adjacent on that vertex. At the end, the subgraph induced by the remaining vertices in the binary heap forms the k-core of the input graph G. The time complexity of this algorithm is $O(|V| + 2|E|lg|V|)$, and space complexity is $O(3|V| + |E|)$. The above algorithm removes all and only those vertices that cannot be part of any output dense subgraphs of minimal size k.

4.3.3 Dense subgraph detection

As shown above, no dense subgraph will span across two different connected components. It suffices to detect dense subgraphs in each individual connected components separately. As for finding dense subgraphs from a given connected component, we present a *Shingling* [Gibson et al., 2005] based heuristic approach which was originally used for detecting web communities. This *Shingling*-based approach can handle large input size and it is also intended for detecting dense subgraphs of arbitrarily size in practice. The basic idea behind this *Shingling* approach is: In a

dense subgraph, each vertex is supposed to be connected to most of the other vertices in the same dense subgraph; thus any two vertices in the same dense subgraph are supposed to share most of neighbors (outlinks). The heuristic we are using is that if two vertices have a high overlap of their outlinks, then most likely they belong to the same dense subgraph. It is a necessary but not sufficient condition to group vertices into the same dense subgraph, and some post-processing check is required to make sure the grouped vertices forms a dense subgraph. In this heuristic approach, the set comparison becomes the kernel problem. Although several optimal algorithms can be applied to perform this comparison, they cannot be scaled to large input size as the required computation is relatively expensive.

For each connected component, we first note that any connected component within G is after all an instance of another undirected graph $C_i(V_{C_i}, E_{C_i})$. And any such C_i can be transformed into an equivalent undirected bipartite graph $B_d(V_1, V_2, E')$ such that $V_1 = V_2 = V_c$ and $E' = \{(i, j), (j, i) \mid (s_i, s_j) \in E_c\}$. The problem of finding dense subgraphs in C_i is equivalent of finding subset pairs $A \subseteq V_1$ and $B \subseteq V_2$ such that A and B are "well" connected to one another and $\frac{|A \cap B|}{|A \cup B|} \geq \tau$ for some cutoff $0 << \tau \leq 1$. The ratio check is sometimes referred to as the Jaccard index and here it is primarily used to ensure that $A \approx B$. Reducing the problem into a bipartite graph problem has a couple of distinct advantages:

1. The problem of finding dense subgraphs in bipartite graphs has been addressed

in the context of finding web communities, and practically efficient approximation strategies exist [Flake et al., 2000, Gibson et al., 2005]. The subtle difference is that for web communities a dense subgraph is a pair of subsets A and B such that the each vertex (or website) in B is pointed by a majority of vertices in A, without necessitating $A \approx B$. In other words, a dense subgraph in B_d as per our definition can be expected to be detected by an algorithm that solves the corresponding web community dense subgraph problem. But not every dense subgraph reported by the latter may satisfy our $A \approx B$ criterion. But this is an added constraint that can be easily tested in a post-processing step.

2. It is possible to generalize the notion of finding dense subgraphs in bipartite graphs and implement other variations of the problem in the context of protein family detection. For example, instead of duplicating V_c on both sides of the bipartite graph, we can copy it to V_2 and assign V_1 to a set of conserved fixed-length words shared by two or more sequences in V_2. Now, finding dense subgraphs in this new bipartite graph (without the Jaccard index check) could lead to the partitioning of protein sequences in V_2 into groups based on multiple conserved domains or motifs. While more variants are possible, we do not consider such variants in this work.

Our algorithm is based on the bipartite graph problem reduction described

above. We first transform each connected component c_i into an equivalent undirected bipartite graph B_d. The next step is to distribute the bipartite graphs among multiple processors in a loading balanced manner, and apply bipartite dense subgraph detection algorithms individually on each connected component. We implemented the *Shingling* algorithm by [Gibson et al., 2005] as it is suited for very large inputs. Before getting down to the detailed algorithm, couple of more notations are given as follows:

Notation: Given a vertex v in an undirected bipartite graph $B_d = (V_1, V_2, E')$, $\Gamma(v)$ denotes the set of its out-links and is given by $\{u \mid (v, u) \in E'\}$. Given parameters (s, c), a "shingle" [Broder et al., 1997] of a vertex v is an arbitrary $s-$element subset of $\Gamma(v)$, and an "$(s, c)-$shingle set" of v is a set of c randomly picked shingles of v.

As shown above, the kernel part of the dense subgraph detection problem is set comparison problem. A brute-force implementation is to compute the set intersection of the two input sets of outlinks. However this could take $O(|\Gamma(v_i)| \times |\Gamma(v_j)|)$; this could be reduced to $O(|\Gamma(v_i)| + |\Gamma(v_j)|)$ by mapping vertices to unique integers. A third, and faster alternative is to perform random *"sampling"*. This is what is done in the "Shingling" method. More specifically, we sample out s-element subset (called "shingle") from the two sets, and if the two sets are similar to each other, then it is more likely that the two sampled shingles are equivalent. A formal theory, called

min-wise independent permutations [Broder et al., 2000] proves that if the two sets being compared intersect to a high degree, then it is very likely to find two identical shingles. However, if two "shingles" are not identical, it does not necessarily imply that the two sets where the two "shingles" are extracted are totally different. To compensate for such misses, the random sampling procedure can be run multiple times. Based on this idea, set comparisons can be efficiently performed on large-scale input without requiring brute-force comparisons. To implement this idea for dense subgraph detection, the outlinks of each vertex is sampled into a set of "shingles". Later these "shingles" serve as the connecting points to group these corresponding vertices together.

Algorithm 6 Shingle (v_i, s, c)

1. P: a 64 bit large primer number
2. $A[1..c]$, $B[1..c]$: random numbers in $[1..P]$
3. H: hash function from a string to a integer
4. $\Gamma(v_i) = (o_1, o_2, \ldots, o_m)$
5. **for** $j = 0$ **to** m **do**
6. $\quad x[i] \leftarrow o_i$
7. **end for**
8. **for** $j = 1$ **to** c **do**
9. $\quad$ **for** $i = 1$ **to** m **do**
10. $\quad\quad y_i \leftarrow (A[j] \times x[i] + B[j]) \bmod P$
11. $\quad$ **end for**
12. $\quad Y \leftarrow \bigcup_{i=1}^{m} \{y_i\}$
13. $\quad (y'_1, y'_2, \ldots, y'_s) \leftarrow$ minimum s elements from $Y = \{y_1, y_2, \ldots, y_m\}$
14. $\quad s_j(v_i) \leftarrow H(\text{“}y'_1 \circ y'_2 \circ \ldots \circ y'_s\text{”})$
15. **end for**
16. $S(v_i) = \bigcup_{j=1}^{c} \{<v_i, s_j(v_i)>\}$

Algorithm 6 presents the details to sample out a set of "shingles" for a vertex. The detailed dense subgraph algorithm can be illustrated in Figure 4.1, and it operates in two-passes:

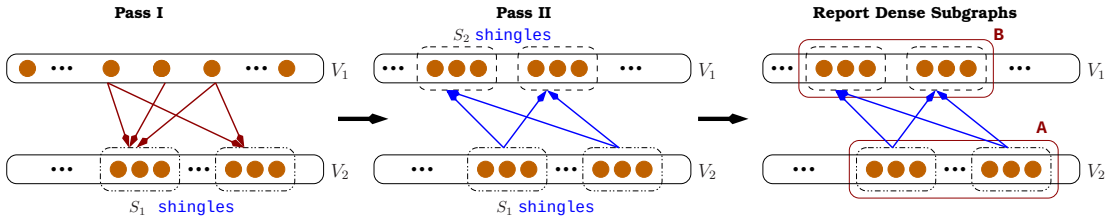

Figure 4.1: Illustration of the two-pass Shingling algorithm. Dotted boxes represent shingles.

i) **Pass I:** An (s, c)-shingle set (denoted by $S(v_i)$) is generated for each vertex $v_i \in V_1$. For ease of implementation, each shingle is mapped to an integer using a string to integer hash function. The results are recorded as a 2-tuple $< s(v_i), v_i >$, where $s(v_i) \in S(v_i)$. Let S_1 denote the set of all shingles generated in this pass. Next, vertices sharing the same shingle are grouped. This is achieved by sorting the tuples based on shingle values. The resulting tuple list is input to the second pass.

ii) **Pass II:** The algorithm reverses direction and generates an (s, c)-shingle set for each first level shingle $s(v_i)$. The result is a set of second level shingles S_2, representing vertices from V_1.

In the final reporting step, all connected components, as defined by S_2 shingle to S_1 shingle edges, are enumerated and their constituent vertices recorded. We implemented this grouping using the union-find data structure [Tarjan, 1975]. Finally, the two vertices subsets that are covered by this connected component are reported as dense subgraphs (provided they satisfy the Jaccard index test as well). For a connected component with $|V_c|$ nodes, our implementation of the Shingling algorithm takes $O(\frac{|V_c| \times c^3}{s^2})$ time and $O(\frac{|V_c| \times c^2}{s})$ space complexities in the worst-case. In practice, the observed costs could be significantly lower and is data dependent.

4.4 Experiments

4.4.1 Implementation

The *pClust* code was implemented in C/MPI, and MPI is primarily used to write the wrapper to distribute the connected components among computation nodes. The default setting for *Shingling* algorithm is: $s = 2$, and $c = 200$. These default parameters for (s, c) should change based on the size of the dense subgraphs we are expecting.

4.4.2 Experimental setup

Input Data: For our experiments, we downloaded an arbitrary subset of predicted protein families from the GOS project such that the sum of the number of sequences in all these families was roughly 2 million. Of these, 755,441 sequences were "redundant" — i.e., reported to be contained in at least one other sequence in the set with 98% identity [Yooseph et al., 2007]. Only the remaining ∼1.24 million non-redundant sequences are used in our experiments. The statistics for the 2M data are shown in Table 4.1.

# Total seqs.	# Non-redun. (nr) seqs	Total residues (nr)	Mean. seq. length$\pm\sigma$ (in res.)
2,004,241	1,248,800	308,432,812	247$\pm$173

Table 4.1: Input sequence statistics for ∼2M sequences extracted from the GOS project database.

Experimental Platform: Due to the randomness and large prime number requirements in Shingling algorithms, all tests were performed on a 24-node Linux commodity cluster with a gigabit ethernet interconnect and each node containing 8 2.33GHz Xeon CPUs and an 8GB RAM.

4.4.3 Parametric studies

Parametric study of shingling algorithm: We conducted experiments to evaluate the fluctuations in quality of the output dense subgraphs due to the parameters (s, c) in the Shingling algorithm. The 20K data (representing 18 predicted families in GOS approach) was used in parametric study for *pClust* approach. First the sequences are passed through *pGraph* code to construct the sequence graph, then all connected components are enumerated as the input for Shingling algorithm. Several (s, c) combinations: $s = [1, \ldots, 10]$ and $c = [20, \ldots, 200]$ were used in our experiments. In the end, the resulting dense subgraphs were compared against the corresponding "benchmark" (predicted 18 protein families in GOS). The results are plotted in figure 4.2. As all the combinations yielded a specificity of 100% against the benchmark, only changes in sensitivity were recorded and shown in the plot.

The larger the value of s, the lesser the probability that two vertices will share a shingle. Alternatively, a smaller value of s is better suited to enhance the chance of detecting not-so-dense subgraphs. The parameter c is intended to create the opposite effect, because it represents the number of random trials performed to test the similarities between two sets of outlinks. Also, it is not computationally practical to exhaustively compare all shingles of each pair of vertices, and the parameter c offers

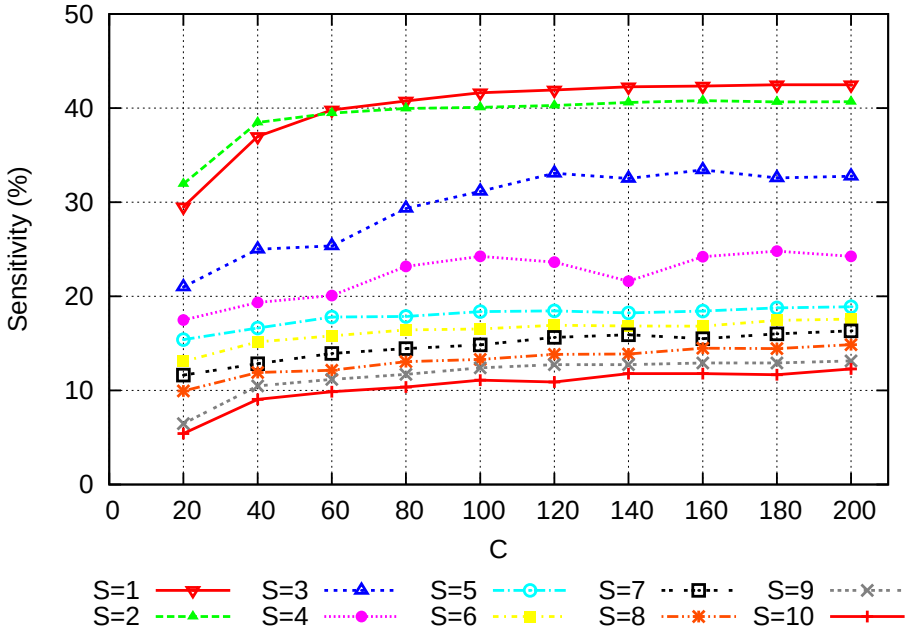

Figure 4.2: Parametric study of the (s, c) parameters on the sensitivity of the Shingling algorithm.

an alternative to restrict this computation space. This is achieved by using the min-wise independent permutation property [Broder et al., 2000]. Even if two vertices share a modest number of out-links, the randomness in this property can be used to increase the probability that such vertex pairs are found. This will be particularly helpful for detecting larger dense subgraphs, as they are expected to be less dense. These parametric controls make the Shingling algorithm an ideal choice for large-scale metagenomic data.

Effect of parameter s: We observed that as s is increased, the sensitivity reduces (for any fixed c), which is consistent with the expectation. s determines the size of a shingle (an s-element subset of out-links of a given node), and if two vertices u and v share a shingle at random they are likely to be contained in the same dense subgraph. Therefore it is natural to expect the sensitivity to reduce as the size of the shingles grows. On the other hand, a low value for s may lead to merging of weakly related sequences and therefore could adversely affect specificity. We however did not observe this adverse effect for the 20K data.

Effect of parameter c: The parameter c is intended to have the opposite effect relative to s — i.e., a larger value of c means more number of trials to find a shared shingle between any two vertices. The Shingling algorithm uses c randomly picked shingles for comparison using the min-wise independent permutation principle. Figure 4.2 confirms this trend although from c values 60 through 200 the rate of improvement in sensitivity is very marginal, implying that for this data a change of c beyond 60 has negligible effect on improving sensitivity. However, for larger data sizes, it may still be preferable to use larger values of c if run-time is still affordable.

4.4.4 Qualitative assessment

First, we devised an experiment to verify the correctness of our sequence homology detection (*pGraph*) code. The expectation is as follows: under identical alignment settings, the homology graph generated by the *pGraph* should match the homology graph generated by a brute-force implementation that performs all-against-all sequence comparison using the Smith-Waterman algorithm. However, the *pGraph* code has a parameter (exact match filter) ψ which is intended to filter out poor quality pairs destined to failed alignment test. Considering the wide variance in the sequence lengths in metagenomic data ([28$\cdots$5,920] in our input) and the low similarity cutoff requirements (e.g. η =40%), it is unlikely but still possible that there are pairs that do not satisfy the ψ exact match criteria for larger values of ψ but succeed in alignment test. To compute the loss in such information, we implemented a brute-force all-against-all sequence comparison code and matched the resulting the homology graph with the homology graph constructed by our *pGraph*. We found that at $\psi = 3$, the homology graph resulted from *pGraph* is over 99.99% identical to the brute-force sequential implementation.

Comparative Evaluation against GOS Predicted Families: Next, we compare our results against the clusters (intermediate results) and the predicted families (final results) generated in the GOS project [Yooseph et al., 2007]. The experiments

were implemented as comparisons of different partitions of the same input ORF data, as schematically illustrated in Figure 4.3. The dense subgraphs reported by *pClust* represents a partition of the n sequences[5], and we call it the "*pClust partition*". The clusters generated in the GOS approach were downloaded from the CAMERA database and are collectively labeled the "*GOS partition*". Finally, the set of predicted families generated by the GOS approach by subsequent expansion of clusters using profile-profile matching represents the third partition. While a true protein family benchmark for this metagenomic data is not yet available, the rigorously followed procedures during cluster expansion and validation makes the set of predicted families output by the GOS team a reasonable benchmark for comparison here. We labeled this set as the "*Benchmark partition*" and compared the other two test partitions against it as described below.

Let s_i and s_j be any two sequences in S. Let $g_p(i)$ denote the group that owns s_i in partition p. To compare a test partition ("t") against the benchmark partition ("b"), we classified every such pair (s_i, s_j) into one of the four classes:

i) True Positive (**TP**): If $g_t(i) = g_t(j)$ and $g_b(i) = g_b(j)$;

ii) False Positive (**FP**): If $g_t(i) = g_t(j)$ and $g_b(i) \neq g_b(j)$;

---

[5]Singleton sequences that do not get reported as part of any dense subgraph are treated as dense subgaphs with size 1.

iii) False Negative (**FN**): If $g_t(i) \neq g_t(j)$ and $g_b(i) = g_b(j)$;

iv) True Negative (**TN**): If $g_t(i) \neq g_t(j)$ and $g_b(i) \neq g_b(j)$.

Using the above measures, we derived the following:

$$\text{Specificity } (\mathbf{SP}) = \frac{\text{TP}}{\text{TP+FP}} \tag{4.1}$$

$$\text{Sensitivity } (\mathbf{SE}) = \frac{\text{TP}}{\text{TP+FN}} \tag{4.2}$$

$$\text{Overlap Quality } (\mathbf{OQ}) = \frac{\text{TP}}{\text{TP+FP+FN}} \tag{4.3}$$

$$\text{Correlation Coefficient } (\mathbf{CC}) = \frac{\text{TP}\times\text{TN-FP}\times\text{FN}}{\sqrt{(\text{TP+FP})\times(\text{TN+FN})\times(\text{TP+FN})\times(\text{TN+FP})}} \tag{4.4}$$

Ideally, SP=SE=OQ=CC=100%.

Approach	SP	SE	OQ	CC	# Aligned pairs	Run-time
pClust, ψ=10	100.00%	20.87%	20.87%	43.58%	1,107,299	12 min
pClust, ψ=8	100.00%	24.30%	24.30%	47.12%	1,774,780	20 min
pClust, ψ=6	100.00%	33.37%	33.37%	55.51%	3,420,653	40 min
pClust, ψ=4	100.00%	40.68%	40.68%	61.55%	13,338,957	3 hr
GOS	100.00%	21.43%	21.43%	44.18%	X	15 hr

Table 4.2: Qualitative comparison of the pClust and GOS partitions against the benchmark for the 20K data. 'X' denotes data not applicable because the GOS approach uses BLAST.

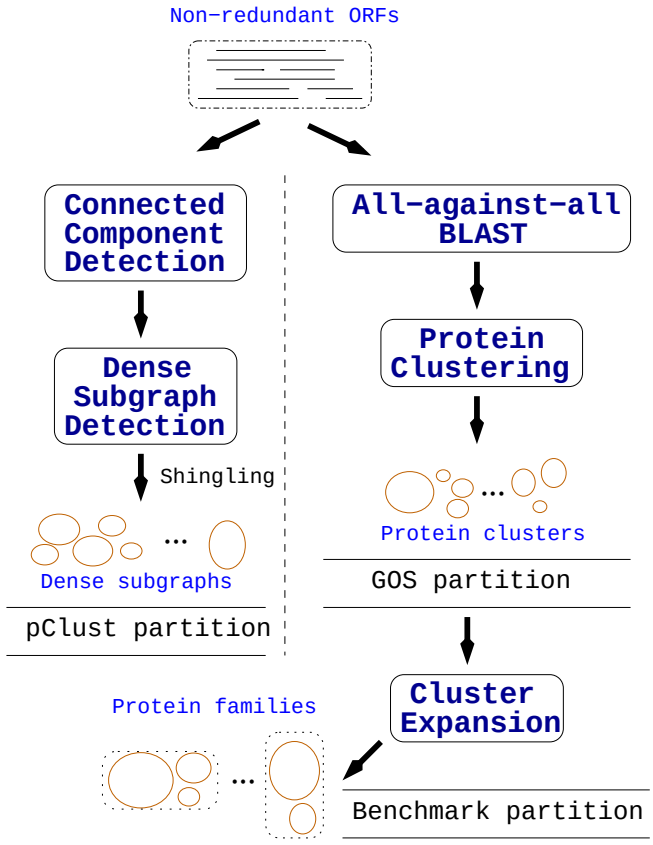

Figure 4.3: An illustration of our strategy for qualitatively comparing the three partitions — *pClust* dense subgraphs, GOS clusters and GOS predicted families.

Experiments on 20K ORFs: We extracted an arbitrary subset containing 20,000 ORFs (representing 18 predicted families) for initial analysis. To enable a direct comparison, the alignment parameters of the GOS cluster creation phase in [Yooseph et al., 2007] were replicated under *pClust*. The parameters for the Shingling algorithm in *pClust* were defaulted to $s = 2$ and $c = 200$ throughout our experiments (after

preliminary parametric studies). Table 4.2 shows the results of comparing the two test partitions against the benchmark. Rows 1 through 4 show the results of *pClust* under different ψ parameter settings. As ψ is decreased from 10 to 4, significantly more pairs of ORFs become eligible for alignment computation, explaining the gradual improvement in sensitivity. At ψ=4, the sensitivity of *pClust* is roughly 1.9X-fold better than the sensitivity of the GOS partition. At $\psi = 10$, the sensitivities of the two partitions become comparable. This implies that it is safe to increase ψ as large as 8 or 10 (in the interest of performance) without degrading quality. The higher sensitivity achieved by the *pClust* approach is a result of two factors: i) the optimal alignment-based evaluation of sequence pairs; and ii) the better approximation achieved by the Shingling algorithm.

On specificity, both partitions generate no false positives. For the GOS partition this is clearly expected because the predicted families are after all direct expansions of the clusters. As for the *pClust* partition, the Shingling algorithm is a different heuristic than the $k-$neighbor heuristic used in the GOS approach. Therefore it could potentially lead to a different grouping. Despite this difference in the underlying methodologies, we observed that *pClust* is able to achieve 100% specificity for this data set.

Table 4.2 also demonstrates the quality-to-performance trade-off. The number of pairwise alignments computed significantly increases even with a little decrease in

# Partition	# Groups	Seqs. included	Largest size	Avg. size
Benchmark	18	20,000	1,111	3,869
GOS	129	12,094	2,469	94
pClust	360	16,288	3,033,	45

Table 4.3: The statistics of different partations of the 20,000 ORFs data set. The "Groups" column corresponds the number of i) predicted protein families in the benchmark; ii) clusters reported by GOS; and iii) dense subgraphs with at least 20 ORFs reported by *pClust*

the value of ψ. However, this significant rise in computation does not necessarily translate to a proportional improvement in quality. This is only expected because more random pairs are likely to share a short exact match by chance, but most such pairs would subsequently fail the alignment test. Furthermore, the dense subgraphs in *pClust* includes 16,288 sequences, which is almost 35% more than that recruited in the GOS clusters (see Table reftab20Kstudy). Recruiting more sequences without affecting the specificity is a significant result.

Experiments on 1.24 million non-redundant ORFs: Next, we present the results of our qualitative assessment on the 1.24 million non-redundant ORFs. The benchmark partition includes all the redundant sequences, and therefore to enable direct comparison of the two partitions (*pClust* and GOS) against the benchmark we

added the redundant sequences back into both partitions — i.e., into the corresponding group of its non-redundant container sequence.

Table 4.4 shows the specificity and sensitivity for both test partitions against the benchmark (rows 1-2). At $\psi = 8$, the *pClust* partition achieves a sensitivity of 17.76%, which is a 1.3X-fold improvement over the GOS partition. For $\psi = 10$, the sensitivities become near identical, consistent with our observations on the 20K data. For $\psi < 8$, the sensitivity is expected to further improve although at the expense of running time. The specificity of *pClust* relative to the benchmark, however, dropped to 97%, and this is due to the differences in the underlying heuristic methodologies used for dense subgraph detection.

To better understand these differences, we also conducted a direct comparison between *pClust* and GOS, treating the *pClust* partition as the "test" and the GOS partition as the "truth" (see rows 3-4 in Table 4.4). We found that the two partitions overlap in only ∼60% of their membership. The results help to quantify the differences introduced by the underlying methodologies: computing alignments differently (dynamic programming vs. BLAST), and using two different heuristics for dense subgraph detection (Shingling vs. k-neighbor). The higher sensitivity (∼86%) suggests that *pClust*'s Shingling heuristic preserves a majority of the cluster memberships. The lower specificity (∼65%) can be attributed to two reasons: i) The *pClust* approach includes significantly more number of sequences than in the GOS clusters (as shown

Approach	SP	SE	OQ	CC
pClust vs. Benchmark	97.17%	17.85%	17.76%	41.56%
GOS vs. Benchmark	100.00%	13.92%	13.92%	37.23%
pClust (size$\geq$ 2) vs. GOS	65.56%	86.53%	59.49%	75.29%
pClust (size$\geq$ 20) vs. GOS	65.60%	86.48%	59.50%	75.28%

Table 4.4: Qualitative comparison of the *pClust* partition (at $\psi = 8$) and the GOS partition against the benchmark for the 1.24 million ORFs. Also shown is a direct comparison of the *pClust* partition against the GOS partition (as the "truth").

in Table 4.5). Therefore, any pair that has at least one ORF that is excluded in the GOS partition will count as a "false positive" in *pClust*; and ii) The GOS approach reports clusters that contain a minimum of 20 members, whereas the corresponding cutoff in *pClust* is 2. This implies that most of the ORF pairs originating from dense subgraphs of size < 20 are likely to be marked as a "false positive" relative to the GOS clusters.

Table 4.5 also shows the differences in the number of sequences recruited by both methods. It can be observed that *pClust* is able to recruit 26% more sequences than GOS for this data set.

In Figure 4.4a we compare the group size distribution in the *pClust* and GOS

Partition	# Groups	# Seqs. included	Group size	
			Largest	Average
Benchmark	813	2,004,241	56,266	2,465±4,372
GOS	6,152	1,236,712	20,027	201±650
pClust (size $\geq$ 20)	6,646	1,414,952	19,066	213±721
pClust (2 $\leq$size$<$ 20)	22,552	148,032	19	7±4

Table 4.5: Table showing the statistics of different partitions for the 1.24 million ORFs, after adding back the redundant sequences.

partitions. As can be observed, both partitions show roughly the same distribution. The *pClust* partition also contains ∼22K "small" dense subgraphs that contain less than 20 ORFs (not shown in plot). Figure 4.5b shows how the sequences included in the individual partitions are distributed among different group size bins. Overall, the *pClust* dense subgraphs cover 871K non-redundant sequences, which is 30.9% more than the number of sequences covered by GOS clusters.

4.4.5 Performance analysis

Table 4.6 shows a summary of the *pClust* run. The connected components detection (CCD) phase partitioned the set of 1.24 million ORFs into 65K connected

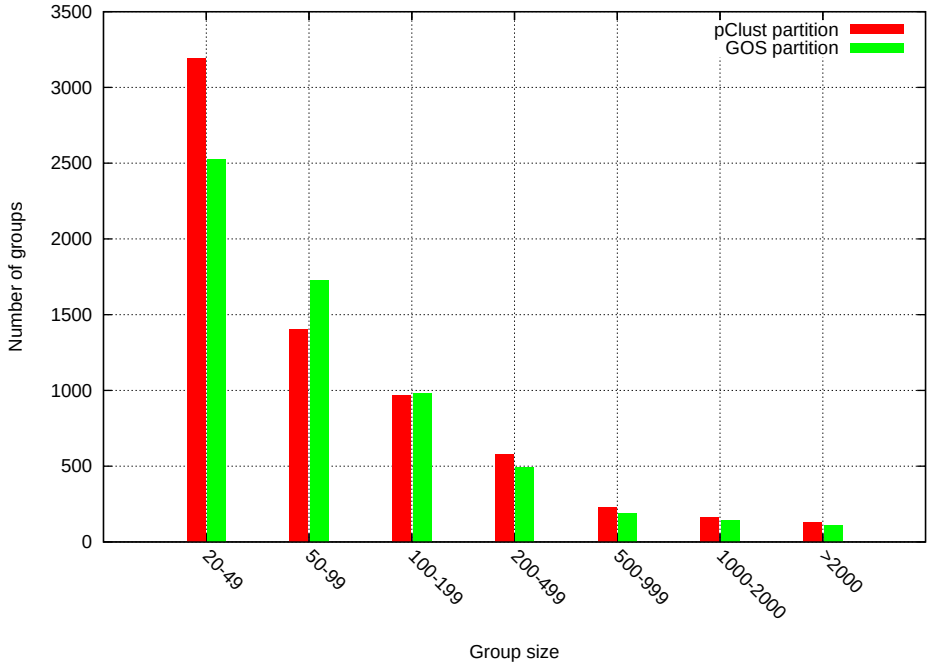

Figure 4.4: Distribution of the dense subgraphs and clusters by their size in the *pClust* and GOS partitions respectively.

components — more than sufficient for a load-balanced distribution among multiple processors for dense subgraph detection. Furthermore, the parallel run-time and memory complexities of the subsequent dense subgraph detection (DSD) phase are directly a function of the size of the largest connected component. The dense subgraph detection (DSD) phase reported 6,646 dense subgraphs of size $\geq$ 20, and the total number of dense subgraphs of size $\geq$ 2 is 29,198; also the largest dense subgraph

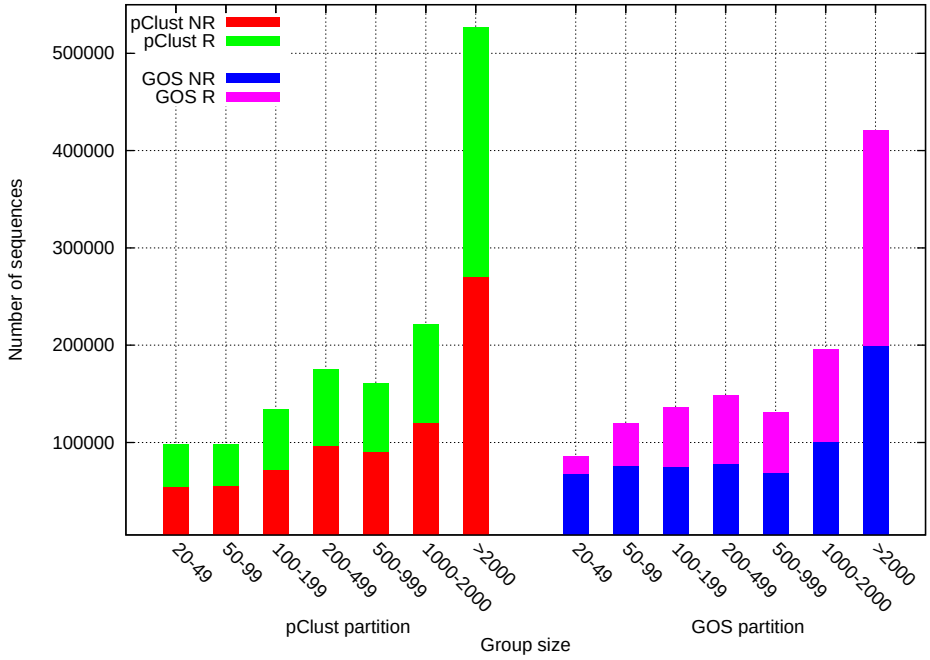

Figure 4.5: Sequence distribution among the different group size bins in *pClust* and GOS partitions. The plot also shows the breakdown between non-redundant ("NR") and redundant ("R") sequences.

reported is of size 19,066. As shown, the largest connected component contained only 10K ORFs, which is less than 1% of the original input and small enough to easily fit inside a single processor's local memory. This allowed us to partition the set of connected components and thereby efficiently parallelize the computation of dense subgraphs as well.

# Input seqs.	Connected Component Detection phase (NR)				
(nr)	# CC	# Seqs included in CC	# Singlets	Avg. CC size	Largest CC size
1,248,800	65,583	973,707	275,093	15	10,707

Table 4.6: Statistics of the connected components (CC) and dense subgraphs (DS) generated by *pClust*.

For exact match length $\psi = 4$ on 2 million ORF sequences, the *pGraph* code finishes in ∼10,000 CPU hours; while for ∼2000 CPU hours are required with $\psi = 8$. In the $\sim$ 2 million sequence, 1.24 million sequences were reported as non-redundant ORFs. A subsequent CCD phase reported 65K connected components, and all these connected components were distributed among 128 Xeon CPUs of the Linux cluster in parallel and were collectively screened for dense subgraphs in ∼30 minutes.

4.5 Conclusion

With the completion of every new metagenomics sequencing project there is a wealth of new genetic information being added to public repositories. Detecting protein families from these sequence data can provide a preview into the functional space of environmental microbial colonies. However, the rate of data accumulation is fast outstripping our ability to analyze them, and therefore it is imperative to present

scalable software solutions to tackle this challenging problem. In this chapter, the sequence clustering problem is transformed into one of the connected component detection and dense subgraph detection problem. For the connected component step, its main task is to break down the large problem instance into subproblems of much smaller size. In this way, the space limitation challenge could be addressed accordingly. Also to efficiently identify dense subgraph on large-scale input graph, we presented a "Shingling" based clustering algorithm. This "Shingling" approach solves the time complexity challenges when applying to large-scale input. To evaluate the effectiveness and accuracy of our approach, we extracted $\sim$2 million sequence from CAMERA portal, and extensive experiments confirmed that our approach is able to improve sensitivity, recruit more sequences, while considerably reducing the time to solution and memory requirements.

CHAPTER 5. CONCLUSION AND FUTURE RESEARCH DIRECTIONS

Metagenomics is an emerging area that is changing the way people are looking at the microbial world, and it is having a profound impact on bioenergy, environmental biotechnology, medicine and agriculture. With the rapid advancement of the high-throughput sequencing technologies, millions of new sequences are produced at an unprecedented rate. Among other interesting computational problems, protein family identification plays the most important role in understanding the structural and functional roles of these new discovered proteins. Although protein family detection problem has been an active research area for the last decade, none of the existing approach can efficiently handle the large volume of data produced in metagenomics. Even those that deploy parallelism resort to brute-force allocation of tasks across multiple computers and to using specialized large-memory high-end platforms for tackling the space problem. It is our strong belief that high performance computing could play a significant role in overcoming this challenge.

In this dissertation, we presented parallel approaches for protein sequence characterization in large-scale metagenomic data sets. Our approaches exploit the aggregate memory and compute power of massively parallel distributed memory su-

percomputers. There are primary two contributions: i) sequence homology graph construction; and ii) protein sequence clustering. In the sequence homology graph construction part, we eliminate the need to do the traditional used all-against-all comparison through an exact-match based filtering technique that uses suffix tree. Experimental results confirmed that 96%-99% workload could be eliminated without compromising the quality of the final result. In sequence clustering part, we transform the sequence clustering problem into one of the connected component detection and subsequent dense subgraph detection problem; the dense subgraph problem is efficiently solved using a "shingling" based heuristic approach, and extensive qualitative analysis demonstrates the fine quality of our clustering approach. Our approach is developed as an open source project, and is available to all communities. Along the line of this dissertation, couple of more research could be explored:

1. **pGraph:** The performance of our current implementation can be further enhanced by augmenting fine-grain parallelism to compute the individual alignments. This can be achieved by substituting the serial alignment code with hardware accelerated alignment computation kernels based on the accelerating platform available at disposal. Such an extension would make the alignment computation much faster and the effect of that along with the possibility of accelerating the pair generation routine needs to be studied in tandem.

The techniques proposed in this chapter could also be extended to other data-intensive scientific applications which are posed with similar challenges in the work generation and work processing. The functions for pair generation at the producer and sequence alignment at the consumer could in principle be substituted with application-specific work generation and processing code (similar to specifying mapper() and reducer() functions in Map Reduce). We plan to incorporate this feature and make it available as a generic parallel library that can be plugged into any other data-intensive scientific computing applications.

2. **pClust:** While run-time is not likely to be an issue for our current dense subgraph detection phase, the peak space requirement $O(\frac{|V| \times c^2}{s})$ (s and c are parameters) has the potential risk to affect the scalability of our clustering stage. In order to overcome this space challenge, some parallelization effort is required for future larger applications. We also noticed that Map/Reduce paradigm could be directly applied to solve this memory challenge. Basically, the mappers are going to emit shingle tuples and the shingles merging process can be achieved through reducers; later the same mapper-and-reducer scheme can be applied again on the previous generated shingle results.

Bibliography

The national center for biotechnology information., 2011. http://www.ncbi.nlm.nih.gov/genbank/.

Swiss institute of bioinformatics., 2011. http://www.uniprot.org/.

454 life sciences, 2011. http://www.454.com/.

S.F. Altschul, W. Gish, W. Miller, and E.W. Myers *et al.* Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

R. Apweiler, A. Bairoch, and C.H. Wu. Protein sequence databases. *Current Opinion in Chemical Biology*, 8(1):76–80, 2004.

A. Bateman, L. Coin, R. Durbin R, and R.D. Finn *et al.* The Pfam protein families database. *Nucleic Acids Research*, 32(D):138–141, 2004.

B. Bollobas. The evolution of sparse graphs. *Graph Theory and Combinatorics*, pages 35–57, 1984.

A. Broder, M. Charikar, A. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60:630–659, 2000.

A. Z. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. *WWW6/Computer Networks*, 29:1157–1166, 1997.

Camera - community cyberinfrastructure for advanced microbial ecology research & analysis., 2011. `http://camera.calit2.net`.

A. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiBLAST. 2003.

A.J. Enright, S. Van Dongen, and S.A. Ouzounis. An efficient algorithm for large-scale detection of protein families. *Nucleic Acids Research*, 30(7):1575–1584, 2002.

G. W. Flake, S. Lawrence, and C. L. Giles. Efficient identification of web communities. In *Proc. ACM SIGKDD*, pages 150–160, 2000.

G. Gallo, M.D. Grigoriadis, and R.E. Tarjan. A fast parametric maximum flow algorithm and applications. 18(1):30–55, 1989.

A. Ghoting and K. Makarychev. Indexing genomic sequences on the IBM Blue Gene. In *Proc. ACM/IEEE conference on Supercomputing*, pages 1–11, 2009.

D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *Proc. VLDB Conference*, pages 721–732, 2005.

S.R. Gill, M. Pop, R.T. DeBoy, and P.B. Eckburg *et al.* Metagenomic analysis of the human distal gut microbiome. *Science*, 312(5778):1355–1359, 2006.

O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, 1982.

J. Handelsman. Metagenomics: Application of genomics to uncultured microorganisms. *Microbiology and Molecular Biology Reviews*, 68(4):669–685, 2004.

J. Handelsman, M.R. Rondon, S.F. Brady, and J. Clardy *et al.* Molecular biological access to the chemistry of unknown soil microbes: a new frontier for natural products. *Chem. Biol.*, 5:R245–R249, 1998.

A. Kalyanaraman, S. Aluru, V. Brendel, and S. Kothari. Space and time efficient parallel algorithms and software for EST clustering. *IEEE Transactions on Parallel and Distributed Systems*, 14(12):1209–1221, 2003a.

A. Kalyanaraman, S. Aluru, S. Kothari, and V. Brendel. Efficient clustering of large EST data sets on parallel computers. *Nucleic Acids Research*, 31(11):2963–2974, 2003b.

A. Kalyanaraman, S.J. Emrich, P.S. Schnable, and S. Aluru. Assembling genomes on large-scale parallel computers. *Journal of Parallel and Distributed Computing*, 67 (12):1240–1255, 2007.

E.V. Kriventseva, M. Biswas, and R. Apweiler. Clustering and analysis of protein families. *Current Opinion in Structural Biology*, 11(3):334–339, 2001.

E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Dover Publications, 2001.

V.M. Markowitz, N.N. Ivanova, and E. Szeto *et al.* IMG/M: a data management and analysis system for metagenomes. *Nucleic Acids Research*, 36(D):534–538, 2008.

E. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

H. Noguchi, J. Park, and T. Takagi. MetaGene: prokaryotic gene finding from environmental genome shotgun sequences. *Nucleic Acids Research*, 34:5623–5630, 2006.

C. Oehmen and J. Nieplocha. ScalaBLAST: A scalable implementation of BLAST for high-performance data-intensive bioinformatics analysis. *IEEE Transactions on Parallel & Distributed Systems*, 17(8):740–749, 2006.

V. Olman, F. Mao, H. Wu, and Y. Xu. A parallel clustering algorithm for very large data sets. *IEEE/ACM Transaction on Computational Biology and Bioinformatics*, 5(2):344–352, 2007.

W.R. Pearson. Searching protein sequence libraries: Comparison of the sensitivity and selectivity of the smith-waterman and FASTA algorithms. *Genomics*, 11(3): 635–650, 1991.

W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, 85(8):2444–2448, 1988.

D.B. Rusch, A.L. Halpern, G. Sutton, and K.B. Heidelberg *et al.* The Sorcerer II Global Ocean Sampling Expedition: Northwest Atlantic through Eastern Tropical Pacific. *PLoS Biology*, 5(3):e77 DOI:10.1371/journal.pbio.0050077, 2007.

F. Sanger, S. Nicklen, and A.R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proc. National Academy of Sciences USA*, 74(12):5463–5467, 1977.

S. Sarkar, T. Majumder, P. Pande, and A. Kalyanaraman. Hardware accelerators for biocomputing: A survey. In *Proc. IEEE International Symposium on Circuits and Systems*, pages 3789–3792, 2010.

C. Schleper, G. Jurgens, and M. Jonuscheit. Genomic studies of uncultivated Archaea. *Nature Reviews Microbiology*, 3:479–488, 2005.

S.C. Schuster. Next-generation sequencing transforms today's biology. *Nature Methods*, 5(1):16–18, 2008.

E.G. Shpaer, M. Robinson, D. Yee, and J.D. Candlin *et al.* Sensitivity and selectivity in protein similarity searches: a comparison of smith-waterman in hardware to blast and fasta. *Genomics*, 38(2):179–191, 1996.

T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

SOLiD. Abi solid sequencing, 2011. `http://www.appliedbiosystems.com/`.

R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

G.W. Tyson, J. Chapman, P. Hugenholtz, and E.E. Allen *et al.* Community structure and metabolism through reconstruction of microbial genomes from the environment. *Nature*, 428:37–43, 2004.

E. Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(1):313–323, 1990.

E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

J.C. Venter, M.D. Adams, E.W. Myers, and P.W. Li *et al.* The sequence of the human genome. *Science*, 291(5507):1304–1351, 2001.

J.C. Venter, K. Remington, J.F. Heidelberg, and A.L. Halpern *et al.* Environmental genome shotgun sequencing of the Sargasso Sea. *Science*, 304(5667):66–74, 2004.

P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

C. Wu and A. Kalyanaraman. An efficient parallel approach for identifying protein families in large-scale metagenomic data sets. In *Proc. ACM/IEEE conference on Supercomputing*, pages 1–10, 2008.

S. Yooseph, G. Sutton, D. B. Rusch, and A. L. Halpern *et al.* The Sorcerer II Global Ocean Sampling Expedition: Expanding the Universe of Protein Families. *PLoS Biology*, 5(3):e16 doi:10.1371/journal.pbio.0050016, 2007.